DCGAN

```
from __future__ import print_function
#%matplotlib inline
import argparse
import os
import random
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML

# Set random seed for reproducibility
manualSeed = 999
#manualSeed = random.randint(1, 10000) # use if you want new results
print("Random Seed: ", manualSeed)
random.seed(manualSeed)
torch.manual_seed(manualSeed)
```

Block 1: In this block, all the needed packages are imported. It is needed to set the *seed* in order to fix the series of random numbers that are going to be generated.

```
# Root directory for dataset
dataroot = "data/celeba"

# Number of workers for dataloader
workers = 2

# Batch size during training
batch_size = 128

# Spatial size of training images. All images will be resized to this
#   size using a transformer.
image_size = 64

# Number of channels in the training images. For color images this is 3
nc = 3

# Size of z latent vector (i.e. size of generator input)
nz = 100

# Size of feature maps in generator
ngf = 64

# Size of feature maps in discriminator
ndf = 64

# Number of training epochs
num_epochs = 5

# Learning rate for optimizers
lr = 0.0002

# Beta1 hyperparam for Adam optimizers
beta1 = 0.5

# Number of GPUs available. Use 0 for CPU mode.
ngpu = 1
```

Block 2: This block contains some of the variables and hyperparameters that are going to be used during the code.

- Data-root: sets the directory where all the data is stored.
- Workers: sets a worker per batch sampler on the RAM in order to process images.
- Batch size: sets the number of images that are going to be processed by the NN at each epoch.
- Image size: Height and width of the input images.
- Nc: number of channels of the input images, for example RGB.
- Nz: Size of the latent vector. This will be a random array of size nc that will be fit on the generator. The generator should be able to fit this "noise" on the data distribution in order to generate images.
- Ngf: Size of the feature maps in the generator.
- Ndf: Size of the feature maps in the discriminator.

In the case of the Discriminator this number will set the dimension of the output from the first convolution. In case of the Generator, will set the dimension of the output from the first Transpose-Convolution.

- Number of epochs: Number of iterations during the training.
- Learning rate: Hyperparameter that will set the step of the optimizer when trying to converge.
- Beta1: Hyperparameter for Adam optimizer.
- Ngpu: number of gpus available

```
# We can use an image folder dataset the way we have it setup.
# Create the dataset
dataset = dset.ImageFolder(root=dataroot,
                           transform=transforms.Compose([
                               transforms.Resize(image_size),
                               transforms.CenterCrop(image_size),
                               transforms.ToTensor(),
                               transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                           ]))
# Create the dataloader
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                         shuffle=True, num_workers=workers)

# Decide which device we want to run on
device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0) else "cpu")

# Plot some training images
real_batch = next(iter(dataloader))
plt.figure(figsize=(8,8))
plt.axis("off")
plt.title("Training Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64], padding=2,
normalize=True).cpu(),(1,2,0)))
```

In this case, as the dataset has a format of multiple images in different folders it is registered with the ImageFolder function. Inside it used the path and a set of transforms that are going to be applied to all the images of the dataset.

Then the dataloader loads the dataset shuffling the images.

```
# custom weights initialization called on netG and netD
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)
```

Weight initialization is implemented for the convolutional layers with a random normal distribution in order to avoid having the same exact numbers on the wights we backpropagation acts. This is an error that happens when all gradients are set to 0 at the beginning. The normal distribution allows to have different starting points for the weights. Not too large, not too small.

This methodology is usually referred as *breaking the symmetry* between weights.

```
# Generator Code

class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d( nz, ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. (ngf*8) x 4 x 4
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size. (ngf*4) x 8 x 8
            nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. (ngf*2) x 16 x 16
            nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # state size. (ngf) x 32 x 32
            nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. (nc) x 64 x 64
        )

    def forward(self, input):
        return self.main(input)
```

The Generator is implemented as a class containing the forward function and the architecture function.

The architecture ins based in 5 blocks:

- 4 blocks containing:
    - Transpose Convolution 2d: These convolutions will upsample the input feature map to a desired dimension of feature maps.
      In the case of the first block, the input is the nz = 100 (latent vector) which will be transformed into a feature map of 64*8=512 feature map.
    - After the convolution, a BatchNorm layer is applied to the result. This will normalize the data by maintaining the mean output close to 0 and the output standard deviation close to 1.
    - Finally, ReLU activation function is used. This function will output the output directly if it is positive, otherwise it will output zero.

- 1 final block with:
    - Transpose Convolution.
    - Tanh activation function, this function will map the data between [-1, 1]

All in all, the Generator takes the latent z vector and maps it into the real distribution data trying to fit the 100 feature dimensions by using the learned weights in order to make them similar to the real data.

The idea is to reduce the upsampled 512 dimensions feature map, by using the transpose convolutions until it fits the 3 channel dimensions, as real RGB images, while making the width and height bigger until reaching the original dataset size.

```
# Create the generator
netG = Generator(ngpu).to(device)

# Handle multi-gpu if desired
if (device.type == 'cuda') and (ngpu > 1):
    netG = nn.DataParallel(netG, list(range(ngpu)))

# Apply the weights_init function to randomly initialize all weights
#  to mean=0, stdev=0.2.
netG.apply(weights_init)

# Print the model
print(netG)
```

After defining the net, we move it to the gpu, initialize the weights for breaking the symmetry, and printing a summary.

```
class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is (nc) x 64 x 64
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf) x 32 x 32
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*2) x 16 x 16
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*4) x 8 x 8
            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*8) x 4 x 4
            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, input):
        return self.main(input)
```

This class is composed by the forward function and the architecture function:

- 4 blocks composed by:
    - Convolutions: that will compress the learned patterns into an increasing feature vector.
    - BatchNorm: As in the generator it will normalize the data.
    - Leacky ReLU: This is the activation function is usually used for nets where we can have spare gradients as GANs.

- 1 block with:
    - Convolution
    - Sigmoig: This activation function will output a probability determining if the image is real or not. (Results between 0 and 1).

In an inverse way, the discriminator will take as an input a batch of images with size 3x64x64 (same output as the generator) and by using Convolutions it will try to learn the pattern that compose the dataset. At every convolution the size of the images bill be reduced but the depth of the feature maps will increase.

```
# Create the Discriminator
netD = Discriminator(ngpu).to(device)

# Handle multi-gpu if desired
if (device.type == 'cuda') and (ngpu > 1):
    netD = nn.DataParallel(netD, list(range(ngpu)))

# Apply the weights_init function to randomly initialize all weights
#  to mean=0, stdev=0.2.
netD.apply(weights_init)

# Print the model
print(netD)
```

As it was done after the design of the generator, now we move the architecture to the gpu and initialize the weights breaking the 0 symmetry. Also, a summary will be printed.

```
# Initialize BCELoss function
criterion = nn.BCELoss()

# Create batch of latent vectors that we will use to visualize
#  the progression of the generator
fixed_noise = torch.randn(64, nz, 1, 1, device=device)

# Establish convention for real and fake labels during training
real_label = 1.
fake_label = 0.

# Setup Adam optimizers for both G and D
optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))
```

Now, the Loss function is defined as criterion = BCELoss() (binary cross entropy function) in order to specify how the *Discriminator* & *Generator* learn.

This function allows us to specify which part of the loss function use with the y input (training data or fake).

A set of 64 vectors with depth nz=100 are generated. These vectors will be fed to the generator in order to map the into the data distribution and transform them into images.

The labels for real and fake data are set to 1 and 0 respectively.

Also, the same Adam optimizer is set for both nets D and G. In this algorithm the momentum is incorporated directly as an estimate of the first-order moment of the gradient. Then it includes bias corrections to the estimates of both, first-order and second-order moments.

```
# Training Loop

# Lists to keep track of progress
img_list = []
G_losses = []
D_losses = []
iters = 0

print("Starting Training Loop...")
# For each epoch
for epoch in range(num_epochs):
    # For each batch in the dataloader
    for i, data in enumerate(dataloader, 0):

        ############################
        # (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))
        ###########################
        ## Train with all-real batch
        netD.zero_grad()
        # Format batch
        real_cpu = data[0].to(device)
        b_size = real_cpu.size(0)
        label = torch.full((b_size,), real_label, dtype=torch.float, device=device)
        # Forward pass real batch through D
        output = netD(real_cpu).view(-1)
        # Calculate loss on all-real batch
        errD_real = criterion(output, label)
        # Calculate gradients for D in backward pass
        errD_real.backward()
        D_x = output.mean().item()

        ## Train with all-fake batch
        # Generate batch of latent vectors
        noise = torch.randn(b_size, nz, 1, 1, device=device)
        # Generate fake image batch with G
        fake = netG(noise)
        label.fill_(fake_label)
        # Classify all fake batch with D
        output = netD(fake.detach()).view(-1)
        # Calculate D's loss on the all-fake batch
        errD_fake = criterion(output, label)
```

```
        # Calculate the gradients for this batch, accumulated (summed) with previous gradients
        errD_fake.backward()
        D_G_z1 = output.mean().item()
        # Compute error of D as sum over the fake and the real batches
        errD = errD_real + errD_fake
        # Update D
        optimizerD.step()
```

Firs we initialize the lists where all loss results are going to be stored.

Then the loop with the number of epochs is created and inside this loop another loop to iterate over the index of the data and data is created too.

In the first part of the training loop, we are trying to train the discriminator. To do so, all gradients are first set to zero, avoiding with this the accumulation of the successive gradients while the backpropagations and the correct parameter update.

Then, we take a batch of real images and set the size for the label vector equal to the size of the batch, this will ensure that for a number of images we will have a label for each image. And then we create that labels vector setting everything as a real label.

After this we feed the *Discriminator* with the batch of real images and we flatten the output with *view(-1)* as the output is a tensor.

With this result we apply the loss function in order to learn that this batch of images was real, so inside the criterion we need to use the *output tensor* and the *labels* that where set to 1 as real images.

And we backpropagate the results in order to calculate the gradients and make the discriminator learn.

After this process of learning what real images look like, we repeat the process for the discriminator, but instead of using real images we use a set of latent vectors that will make understand the net what a fake image is. To do so, it is also needed to be change the labels vector before created to fake images labels.

Finally, in this part of the discriminator's training the error of both real and fake learnings is calculated, and we set the next step for the optimizer.

```
############################
# (2) Update G network: maximize log(D(G(z)))
###########################
netG.zero_grad()
label.fill_(real_label)  # fake labels are real for generator cost
# Since we just updated D, perform another forward pass of all-fake batch through D
output = netD(fake).view(-1)
# Calculate G's loss based on this output
errG = criterion(output, label)
# Calculate gradients for G
errG.backward()
D_G_z2 = output.mean().item()
# Update G
optimizerG.step()

# Output training stats
if i % 50 == 0:
    print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x): %.4f\tD(G(z)): %.4f / %.4f'
          % (epoch, num_epochs, i, len(dataloader),
             errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))

# Save Losses for plotting later
G_losses.append(errG.item())
D_losses.append(errD.item())

# Check how the generator is doing by saving G's output on fixed_noise
if (iters % 500 == 0) or ((epoch == num_epochs-1) and (i == len(dataloader)-1)):
    with torch.no_grad():
        fake = netG(fixed_noise).detach().cpu()
    img_list.append(vutils.make_grid(fake, padding=2, normalize=True))

iters += 1
```

In this part of the loop the *Generator* will be trained.

First of all, all gradients are set to 0 for the *generator* net.

The labels vector is changed again to *real images* as we want to make the generator *think* they are real. This will be tone by making another pass through the discriminator with the fake batch of images and calculating the loss function with this output and the *real images* labels. Then, this result will be backpropagated through the *Generator*, and make the optimizer compute a step.

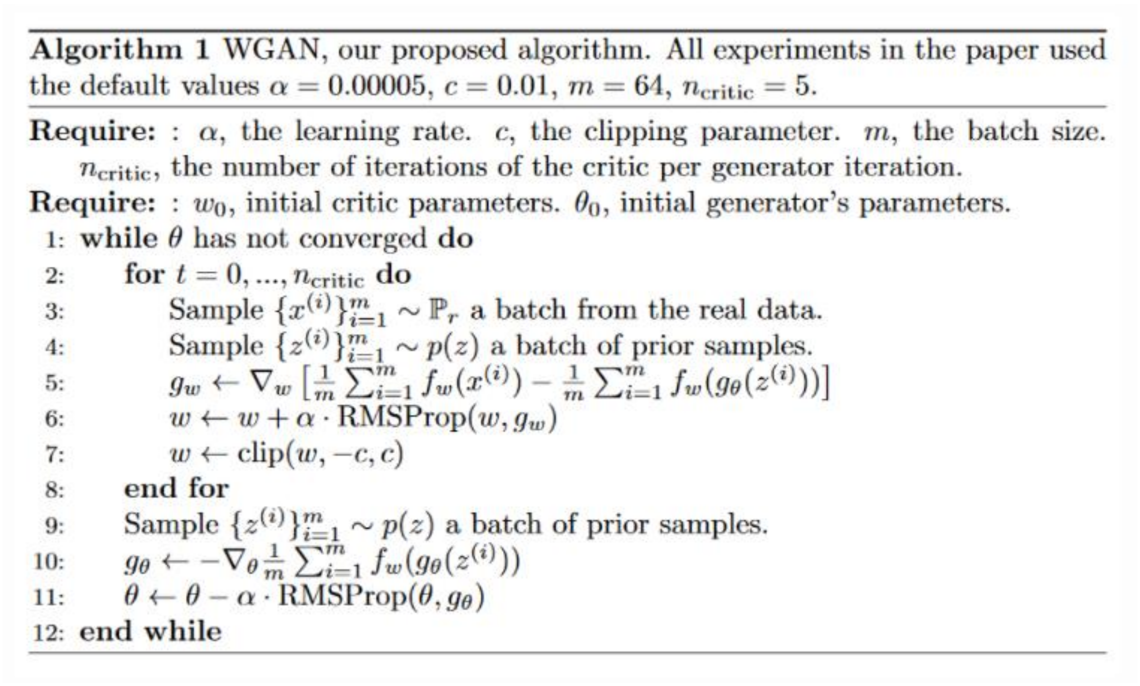
The losses will be stored on the correct list and the iterations number will be updates in order to repeat the loop.

WGAN

In order to implement WGAN some changes should be done to DCGAN, but most of the code will be recycled.

In this image we can observe some of the changes implemented in WGAN:

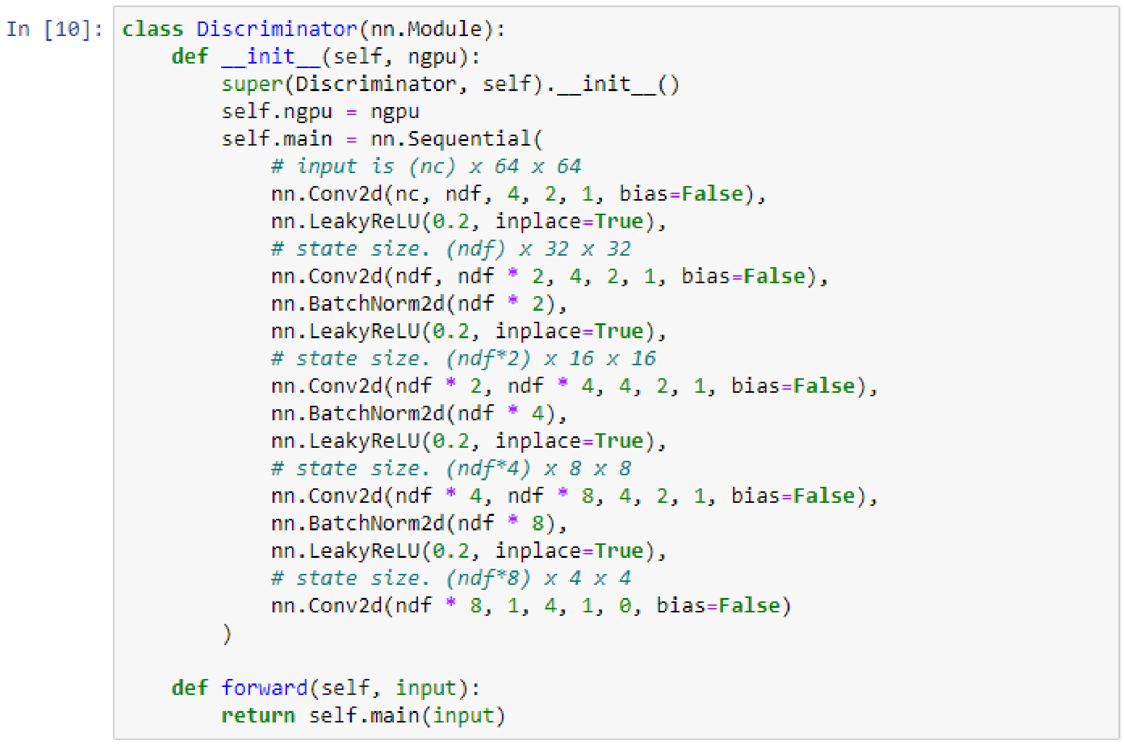
- Hyperparameters:
    - Learning rate 5e-5
    - Batch size = 64
    - Number of iterations for the Critic for an iterator on the generator 5
    - Weight clipping 0.01



Algorithm 1 WGAN, our proposed algorithm. All experiments in the paper used the default values $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{\text{critic}} = 5$.

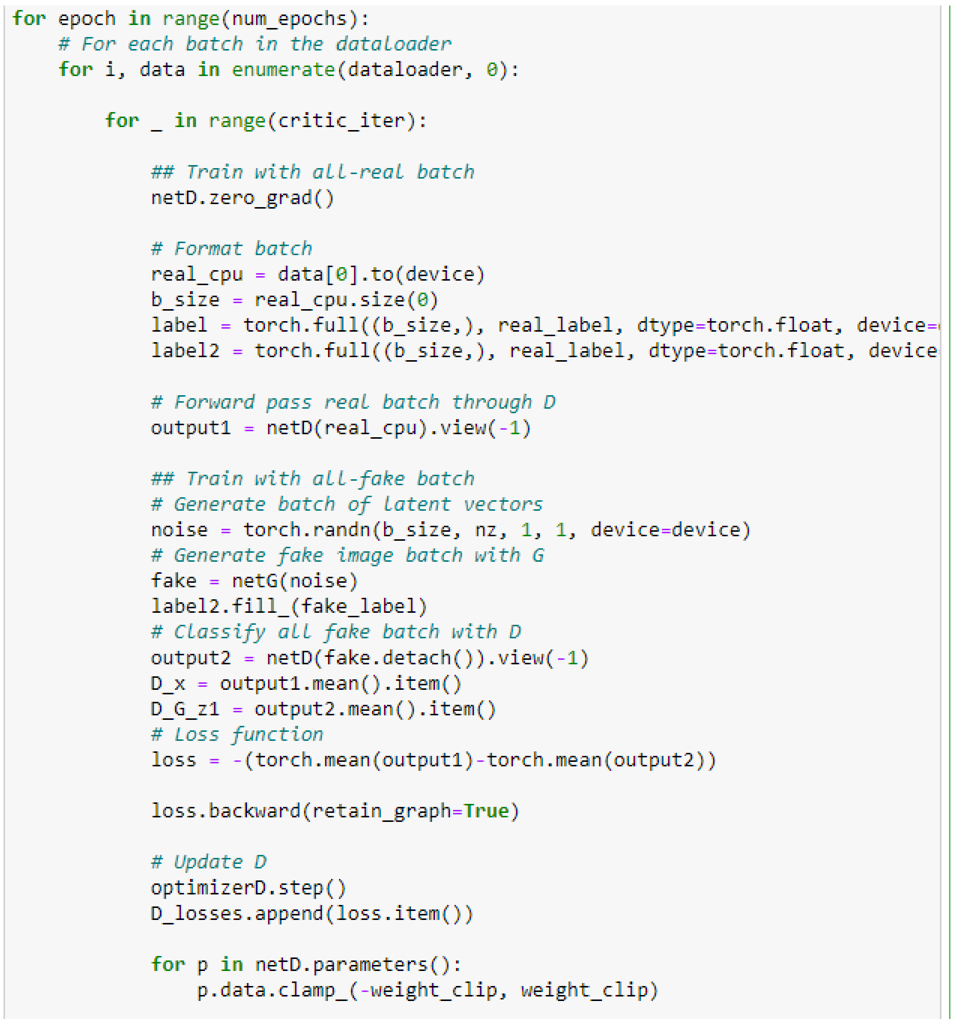
```
Require: : $\alpha$, the learning rate. $c$, the clipping parameter. $m$, the batch size.
    $n_{\text{critic}}$, the number of iterations of the critic per generator iteration.
Require: : $w_0$, initial critic parameters. $\theta_0$, initial generator's parameters.
 1: while $\theta$ has not converged do
 2:     for $t = 0, ..., n_{\text{critic}}$ do
 3:         Sample $\{x^{(i)}\}_{i=1}^m \sim \mathbb{P}_r$ a batch from the real data.
 4:         Sample $\{z^{(i)}\}_{i=1}^m \sim p(z)$ a batch of prior samples.
 5:         $g_w \leftarrow \nabla_w \left[\frac{1}{m}\sum_{i=1}^m f_w(x^{(i)}) - \frac{1}{m}\sum_{i=1}^m f_w(g_\theta(z^{(i)}))\right]$
 6:         $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$
 7:         $w \leftarrow \text{clip}(w, -c, c)$
 8:     end for
 9:     Sample $\{z^{(i)}\}_{i=1}^m \sim p(z)$ a batch of prior samples.
10:     $g_\theta \leftarrow -\nabla_\theta \frac{1}{m}\sum_{i=1}^m f_w(g_\theta(z^{(i)}))$
11:     $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$
12: end while
```

After this, a change on the architecture of the Discriminator should be applied. The last layer, which in DCGAN was a Sigmoid activation function computing the probability of being a certain batch real or fake, now is removed, being the architecture:



```
In [10]: class Discriminator(nn.Module):
             def __init__(self, ngpu):
                 super(Discriminator, self).__init__()
                 self.ngpu = ngpu
                 self.main = nn.Sequential(
                     # input is (nc) x 64 x 64
                     nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
                     nn.LeakyReLU(0.2, inplace=True),
                     # state size. (ndf) x 32 x 32
                     nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
                     nn.BatchNorm2d(ndf * 2),
                     nn.LeakyReLU(0.2, inplace=True),
                     # state size. (ndf*2) x 16 x 16
                     nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
                     nn.BatchNorm2d(ndf * 4),
                     nn.LeakyReLU(0.2, inplace=True),
                     # state size. (ndf*4) x 8 x 8
                     nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
                     nn.BatchNorm2d(ndf * 8),
                     nn.LeakyReLU(0.2, inplace=True),
                     # state size. (ndf*8) x 4 x 4
                     nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False)
                 )

             def forward(self, input):
                 return self.main(input)
```

Now the Training Loop:



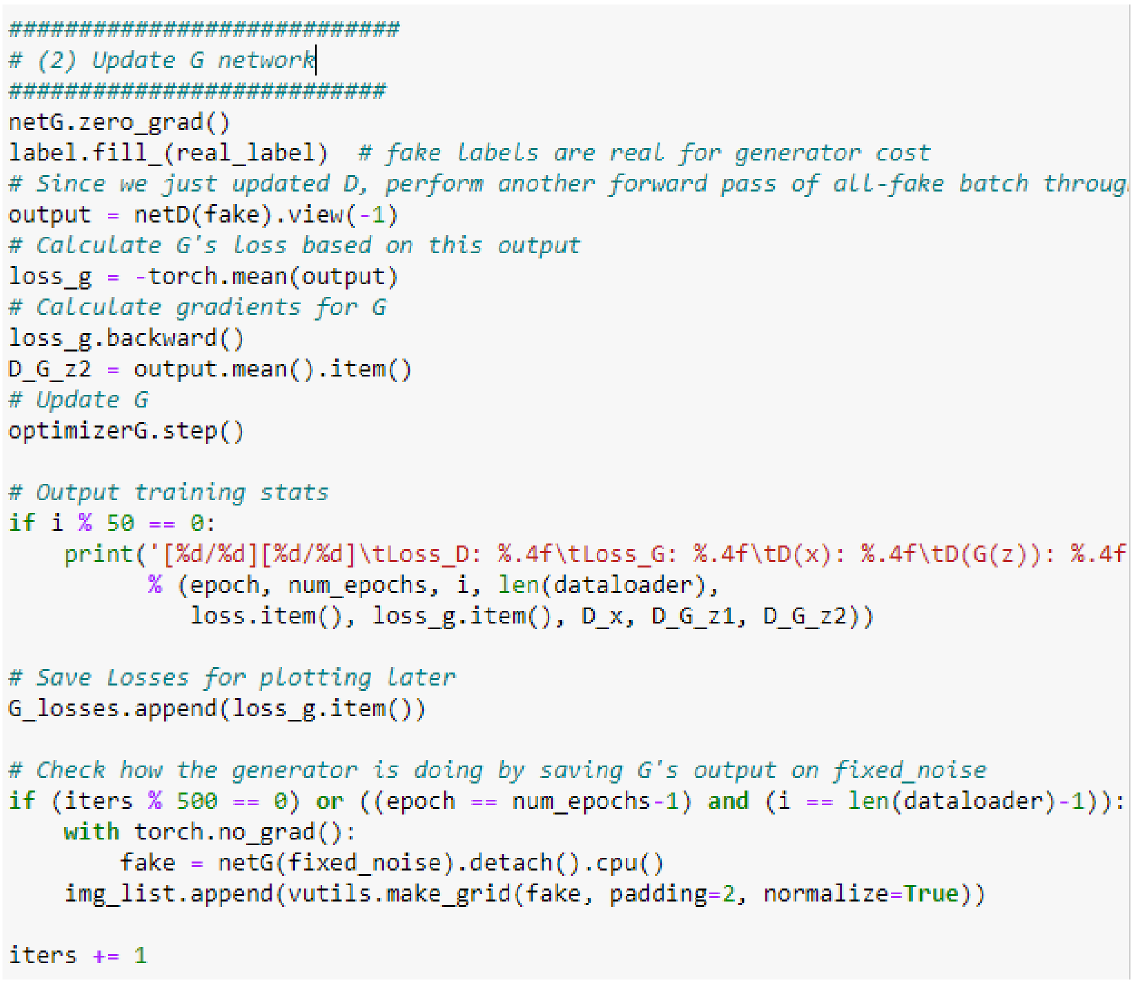
```
for epoch in range(num_epochs):
    # For each batch in the dataloader
    for i, data in enumerate(dataloader, 0):

        for _ in range(critic_iter):

            ## Train with all-real batch
            netD.zero_grad()

            # Format batch
            real_cpu = data[0].to(device)
            b_size = real_cpu.size(0)
            label = torch.full((b_size,), real_label, dtype=torch.float, device=
            label2 = torch.full((b_size,), real_label, dtype=torch.float, device

            # Forward pass real batch through D
            output1 = netD(real_cpu).view(-1)

            ## Train with all-fake batch
            # Generate batch of latent vectors
            noise = torch.randn(b_size, nz, 1, 1, device=device)
            # Generate fake image batch with G
            fake = netG(noise)
            label2.fill_(fake_label)
            # Classify all fake batch with D
            output2 = netD(fake.detach()).view(-1)
            D_x = output1.mean().item()
            D_G_z1 = output2.mean().item()
            # Loss function
            loss = -(torch.mean(output1)-torch.mean(output2))

            loss.backward(retain_graph=True)

            # Update D
            optimizerD.step()
            D_losses.append(loss.item())

            for p in netD.parameters():
                p.data.clamp_(-weight_clip, weight_clip)
```

In this first part at first is taken a batch of real data and pass it through the Critic/Discriminator. Then a batch (same size as real batch) of noise vectors (fake data) is passed through the network.

In order to make the discriminator learn we want to maximize the distance between these two batches of data. Once this is done, we backpropagate and make the step of the optimizer.

Finally, in this part of the loop weight clipping is applied to prevent vanishing gradients while training.

Training the generator:

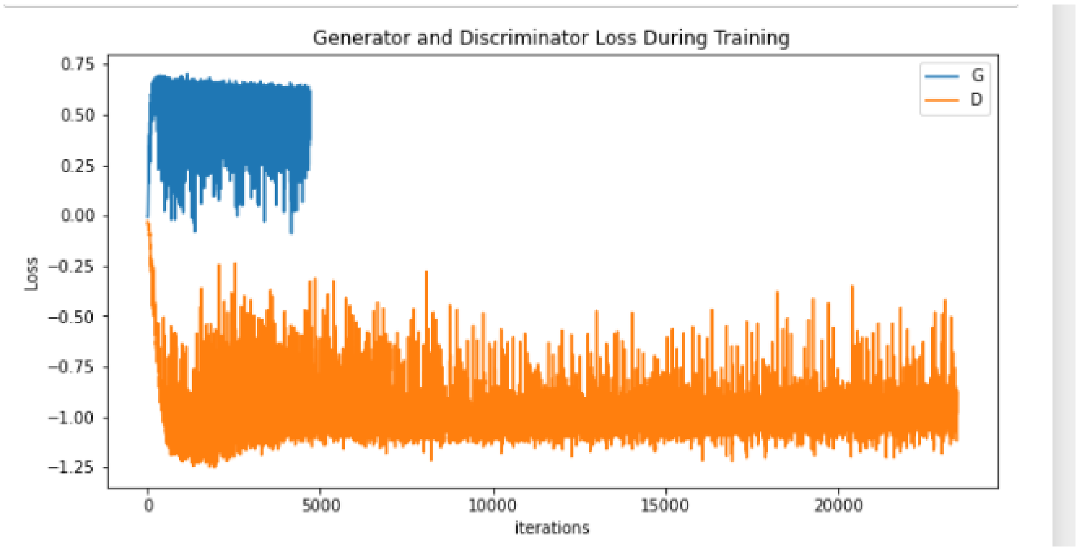


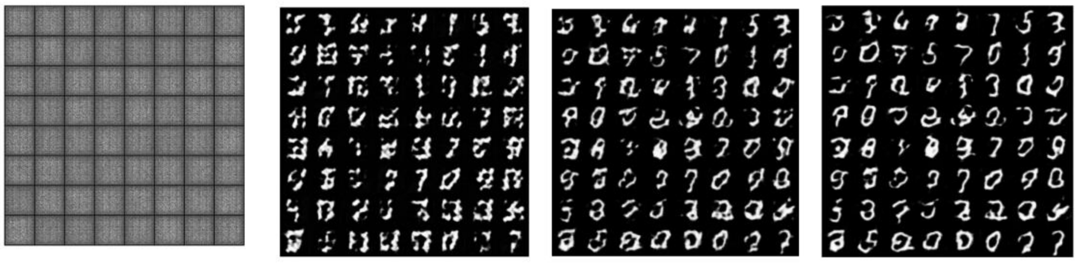
```
############################
# (2) Update G network
###########################
netG.zero_grad()
label.fill_(real_label)  # fake labels are real for generator cost
# Since we just updated D, perform another forward pass of all-fake batch throug
output = netD(fake).view(-1)
# Calculate G's loss based on this output
loss_g = -torch.mean(output)
# Calculate gradients for G
loss_g.backward()
D_G_z2 = output.mean().item()
# Update G
optimizerG.step()

# Output training stats
if i % 50 == 0:
    print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x): %.4f\tD(G(z)): %.4f
          % (epoch, num_epochs, i, len(dataloader),
             loss.item(), loss_g.item(), D_x, D_G_z1, D_G_z2))

# Save Losses for plotting later
G_losses.append(loss_g.item())

# Check how the generator is doing by saving G's output on fixed_noise
if (iters % 500 == 0) or ((epoch == num_epochs-1) and (i == len(dataloader)-1)):
    with torch.no_grad():
        fake = netG(fixed_noise).detach().cpu()
    img_list.append(vutils.make_grid(fake, padding=2, normalize=True))

iters += 1
```

Here we want to minimize the error of the fake data of the discriminator, therefore some output from the discriminator is generated with fake data and then the mean of this output is calculated and backpropagated on the Generator in order to make it learn that this is not the data we want to generate.

After this, a step on optimizer and some prints of losses are generated in order to see the progress.

Results:

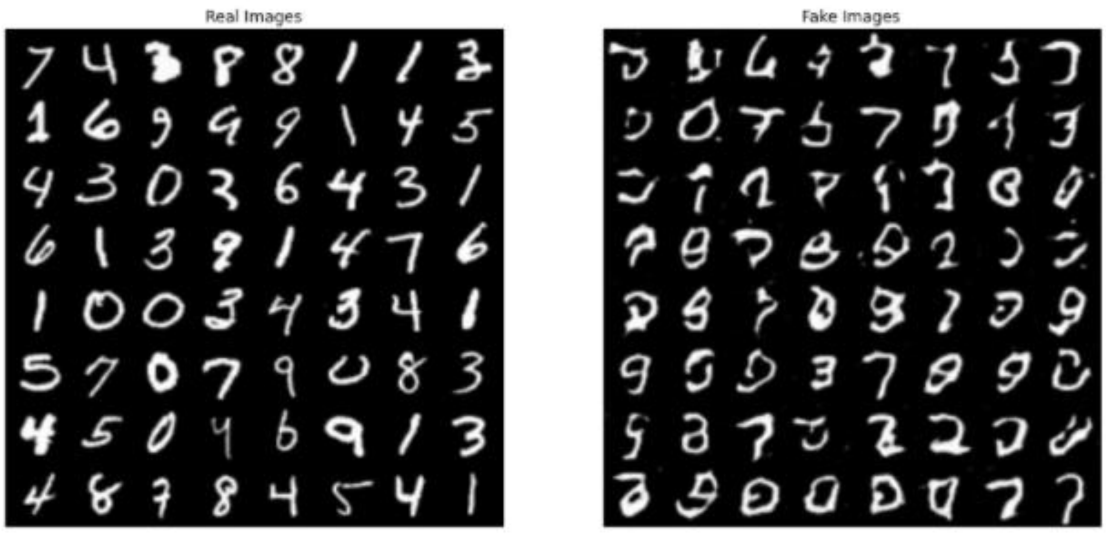




……

We can observe that in the very first iterations the loss function is close to 0 and therefore the images generated are pure noise, but once there is a decay on the loss function of the discriminator it seems that is actually trying to converge and starting to generate similar images to the real ones.

Comparison between real and fake images:



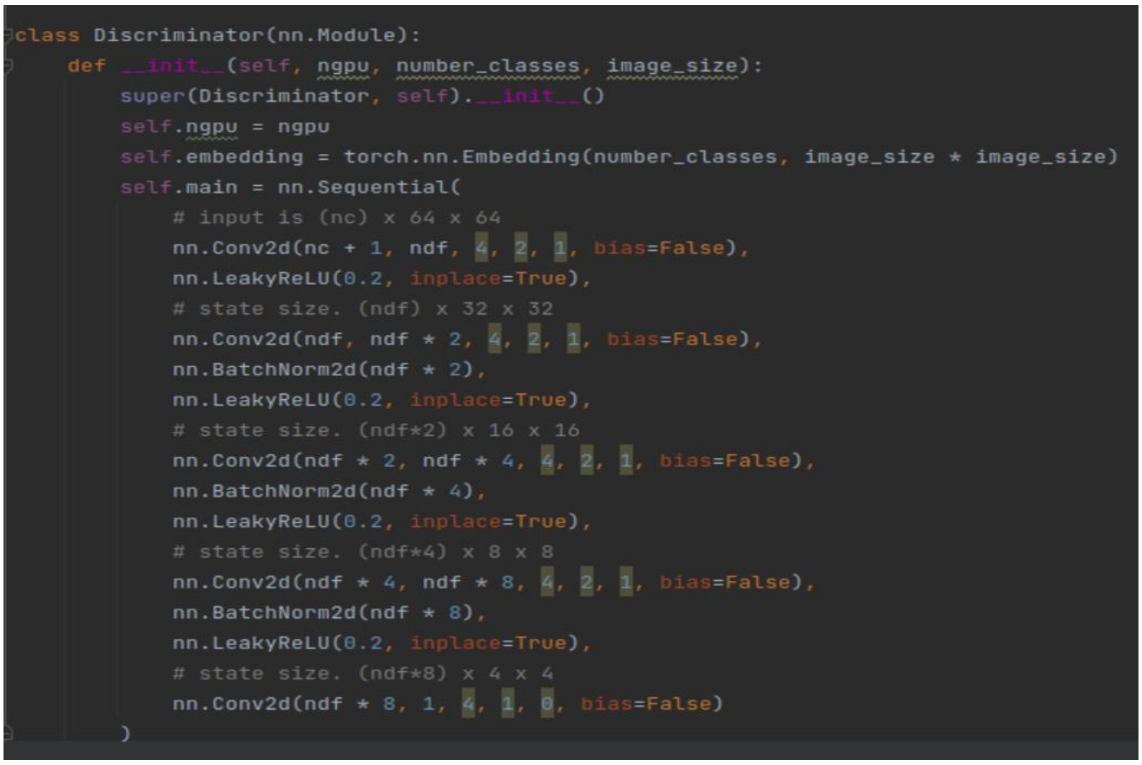
C-WGAN

To convert Wasserstein GAN into a conditional GAN some changes should be applied to WGAN code.

In this GAN the objective is to generate images conditioned by a chosen label. This will be possible by modifying the net in order to introduce the labels while learning.

To do so, it is needed to create embeddings where the labels will be stored and concatenate them in order to add them like a new channel. We should take into account the dimensions of the image embeddings to create the label embeddings in order to set the training data properly.

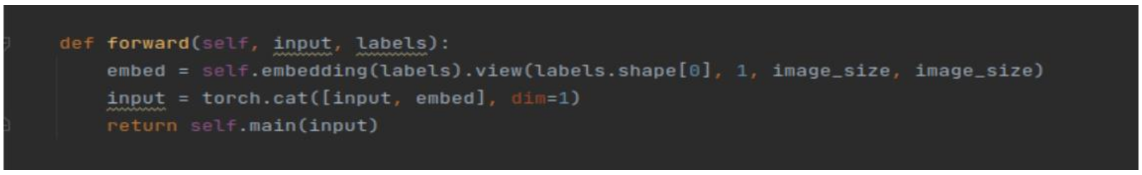
Changes on the Discriminator:



```
class Discriminator(nn.Module):
    def __init__(self, ngpu, number_classes, image_size):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        self.embedding = torch.nn.Embedding(number_classes, image_size * image_size)
        self.main = nn.Sequential(
            # input is (nc) x 64 x 64
            nn.Conv2d(nc + 1, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf) x 32 x 32
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*2) x 16 x 16
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*4) x 8 x 8
            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*8) x 4 x 4
            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False)
        )
```

In the case of the discriminator, we create an embedding with the 1st dimension corresponding to the number of classes in the data, and the second dimension with the total image size (in this case 64x64=4096). Furthermore, these new variables introduced on the network should be added as input variables.

We should modify the forward step also. Now we will add the labels as input variable to the function and we will project those labels into the embedding space previously created.



```
    def forward(self, input, labels):
        embed = self.embedding(labels).view(labels.shape[0], 1, image_size, image_size)
        input = torch.cat([input, embed], dim=1)
        return self.main(input)
```

To properly project the labels a change on the tensor shape should be compute. The variable *"embed"* will contain in its dimensions:

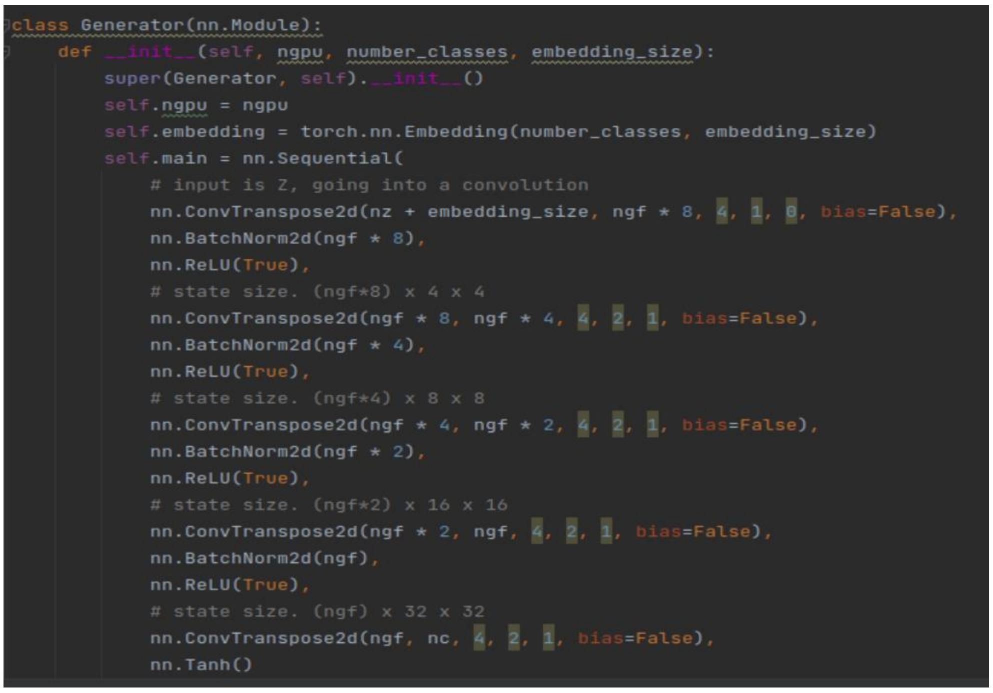
- Label's information.
- Depth (set to one, as we want to add it as a new single channel)
- Width
- Height

Once this is done, we concatenate the label embedding with the image embedding as an extra channel to the images and we pass it through the network.

Finally, to make the network understand that we have an extra channel on the input we add to the first layer a *"+1"* on the input for the first channel.

<u>*Changes on the Generator:*</u>

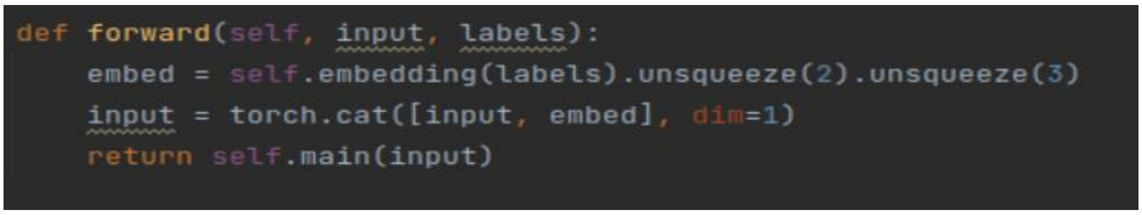
Similar to the discriminator, the main idea is add to the input noise tensor an embedding with the labels.



```
class Generator(nn.Module):
    def __init__(self, ngpu, number_classes, embedding_size):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.embedding = torch.nn.Embedding(number_classes, embedding_size)
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d(nz + embedding_size, ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. (ngf*8) x 4 x 4
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size. (ngf*4) x 8 x 8
            nn.ConvTranspose2d(ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. (ngf*2) x 16 x 16
            nn.ConvTranspose2d(ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # state size. (ngf) x 32 x 32
            nn.ConvTranspose2d(ngf, nc, 4, 2, 1, bias=False),
            nn.Tanh()
```

The embedding will have the dimensions of the number of classes (as before) and the size of the latent noise.

Now, following the steps done on the discriminator, we add the labels as an input variable on the forward step function.



```
def forward(self, input, labels):
    embed = self.embedding(labels).unsqueeze(2).unsqueeze(3)
    input = torch.cat([input, embed], dim=1)
    return self.main(input)
```

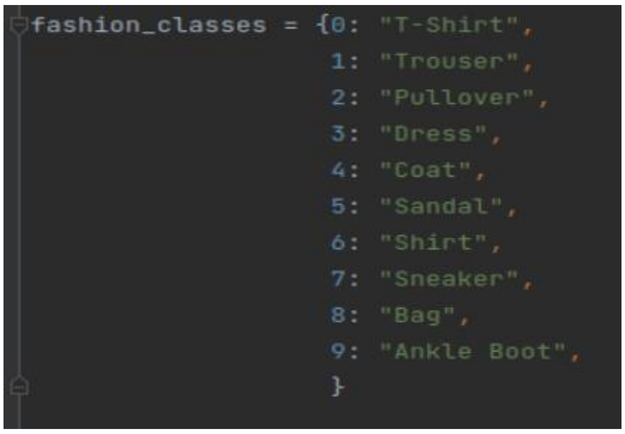
Then, the embedding will be feed with the labels and the shape will be changed to be similar to the input latent noise vector (labels, 1, 1, 1). After this, we concatenate the tensor to the input latent noise vector and pass it through the network.

Similar to the discriminator we should add the embedding label size to the input channels on the first layer to make the net understand the dimensions of the input tensor.

<u>Changes on General Code and Training Loop:</u>

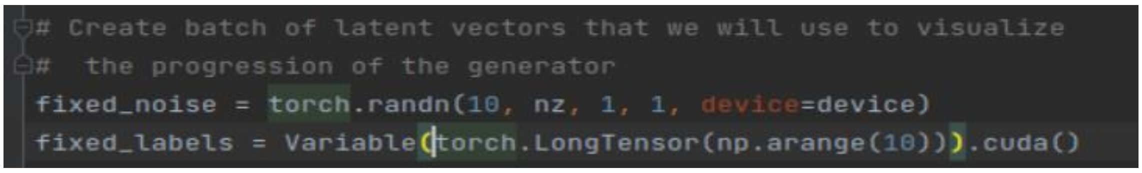
After adapting the architecture of the networks and their forward steps it is necessary to adapt the training loop, where now we will take into account the labels of the dataset.

To so, the first step is to initialize a dictionary where each label will be assigned to a concrete number.



```
fashion_classes = {0: "T-Shirt",
                   1: "Trouser",
                   2: "Pullover",
                   3: "Dress",
                   4: "Coat",
                   5: "Sandal",
                   6: "Shirt",
                   7: "Sneaker",
                   8: "Bag",
                   9: "Ankle Boot",
                   }
```

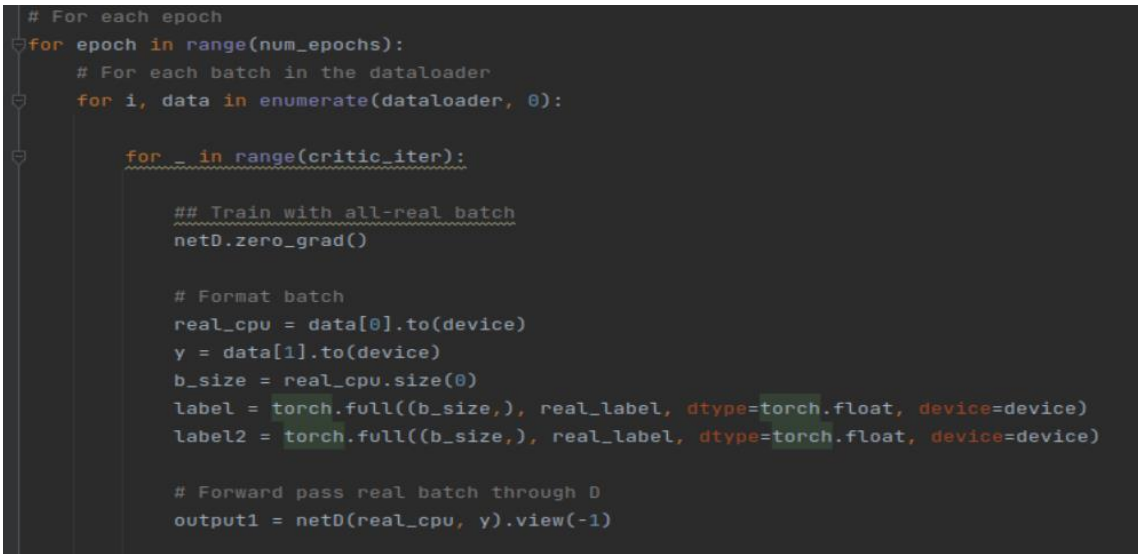
This will be useful to conditionate the output of the GAN with concrete labels stored on a fixed label tensor. Following the example of the fixed noise used to see how the training of the network evolves we will use the fixed labels to push the network to generate the images we want.



```python
# Create batch of latent vectors that we will use to visualize
#  the progression of the generator
fixed_noise = torch.randn(10, nz, 1, 1, device=device)
fixed_labels = Variable(torch.LongTensor(np.arange(10))).cuda()
```

Finally, we include the labels of the dataset into the training loop by creating a batch of labels according to the batch of images that we will feed to the network.

<u>Discriminator:</u>

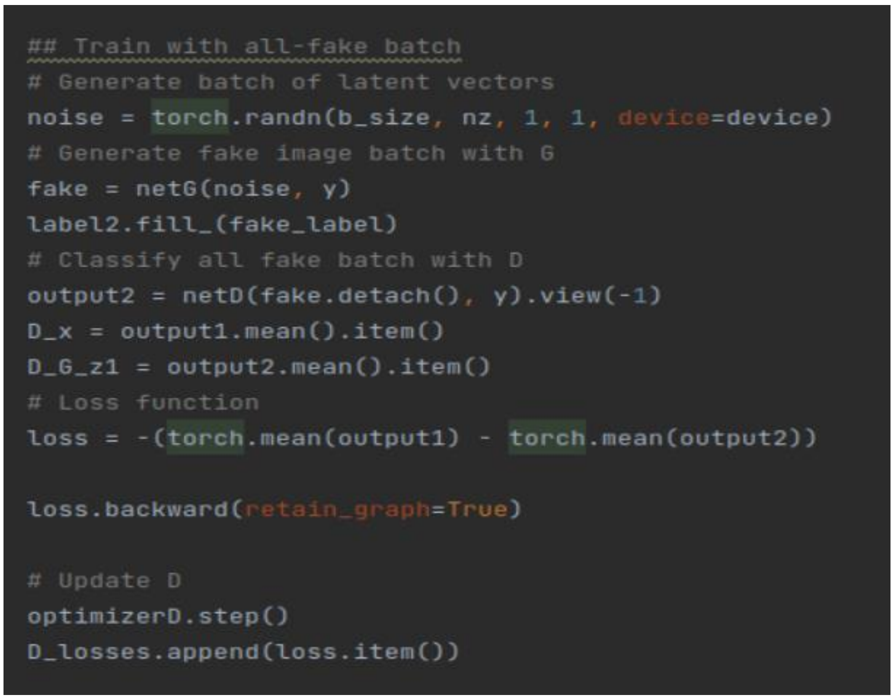


```python
# For each epoch
for epoch in range(num_epochs):
    # For each batch in the dataloader
    for i, data in enumerate(dataloader, 0):

        for _ in range(critic_iter):

            ## Train with all-real batch
            netD.zero_grad()

            # Format batch
            real_cpu = data[0].to(device)
            y = data[1].to(device)
            b_size = real_cpu.size(0)
            label = torch.full((b_size,), real_label, dtype=torch.float, device=device)
            label2 = torch.full((b_size,), real_label, dtype=torch.float, device=device)

            # Forward pass real batch through D
            output1 = netD(real_cpu, y).view(-1)
```

As we can observe, the variable y is where all the labels of the real batch will be stored and then by passing it through the network concatenated as an extra channel of the images.

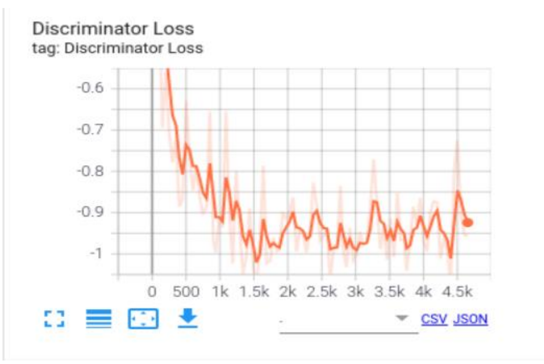
Generator:

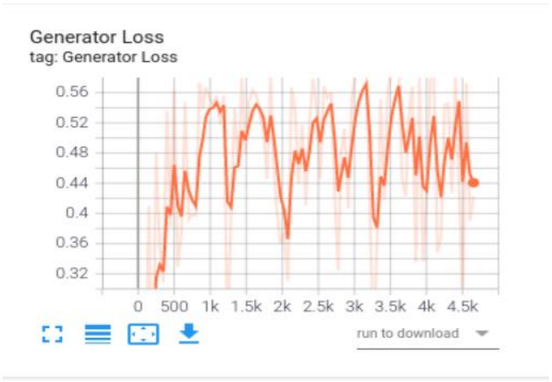
We should feed the Generator with the labels too, in order to make the net learn how to generate images depending on the label.

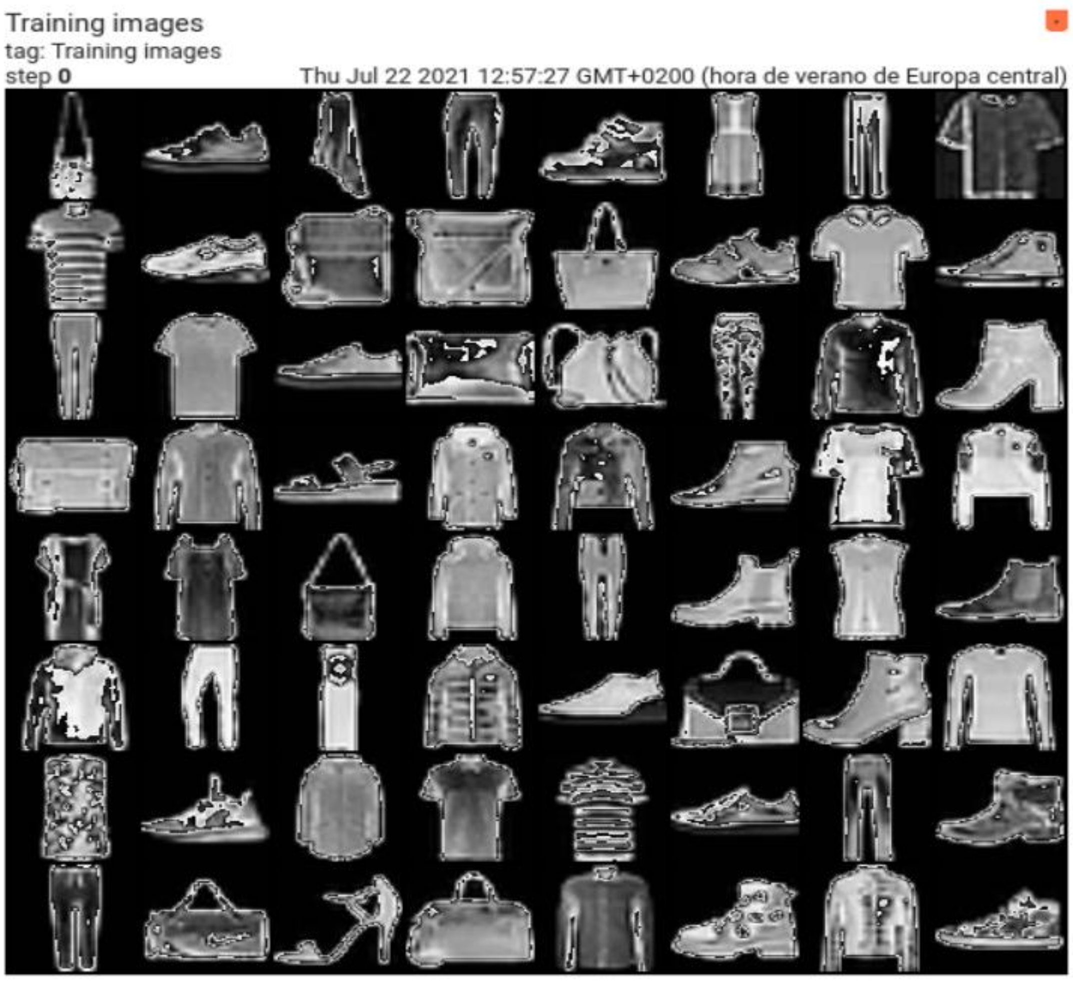


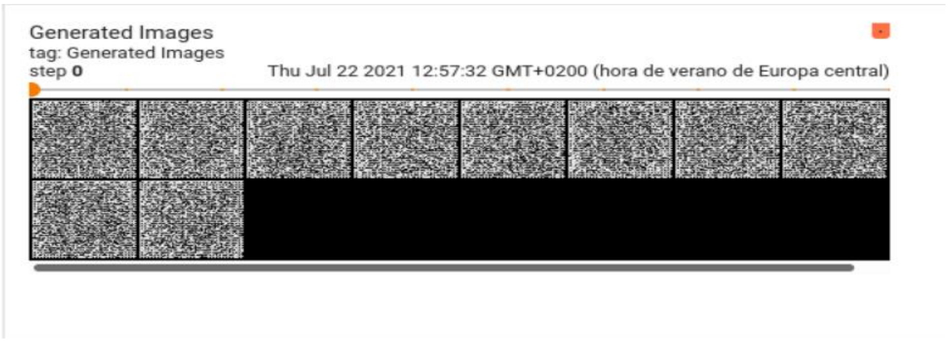
```
## Train with all-fake batch
# Generate batch of latent vectors
noise = torch.randn(b_size, nz, 1, 1, device=device)
# Generate fake image batch with G
fake = netG(noise, y)
label2.fill_(fake_label)
# Classify all fake batch with D
output2 = netD(fake.detach(), y).view(-1)
D_x = output1.mean().item()
D_G_z1 = output2.mean().item()
# Loss function
loss = -(torch.mean(output1) - torch.mean(output2))

loss.backward(retain_graph=True)

# Update D
optimizerD.step()
D_losses.append(loss.item())
```

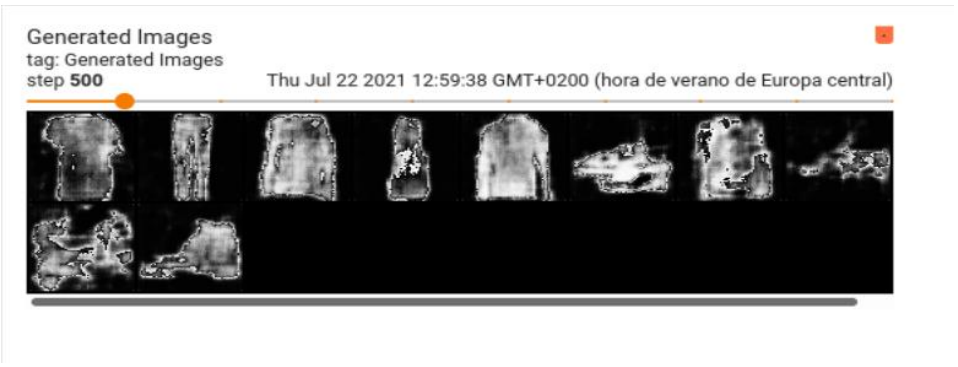
Results:

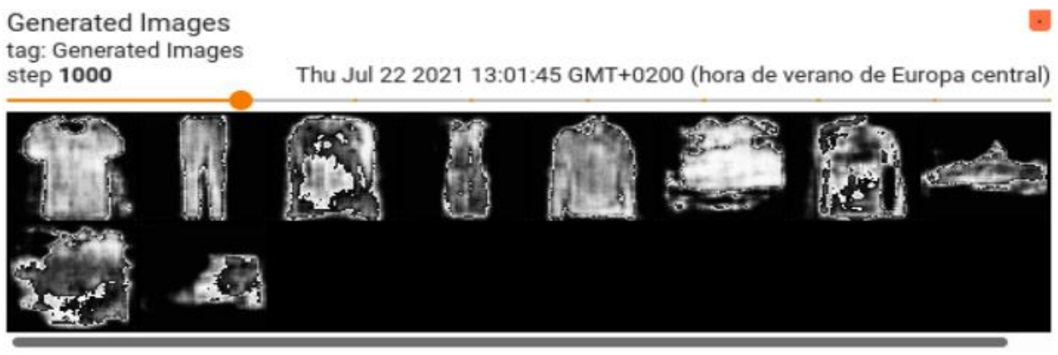


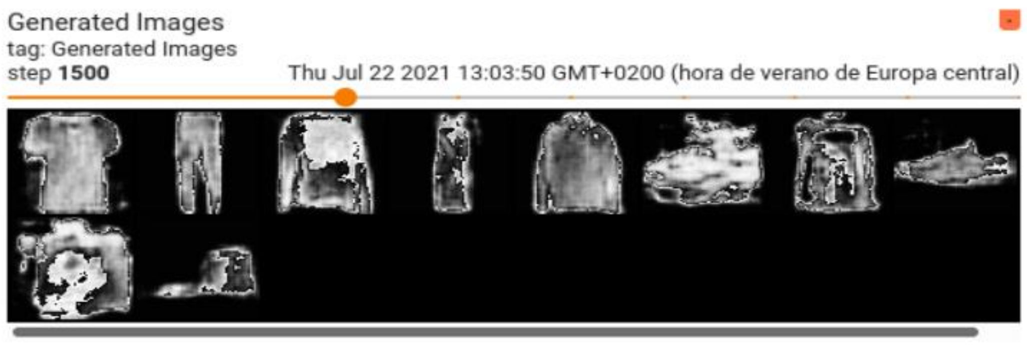


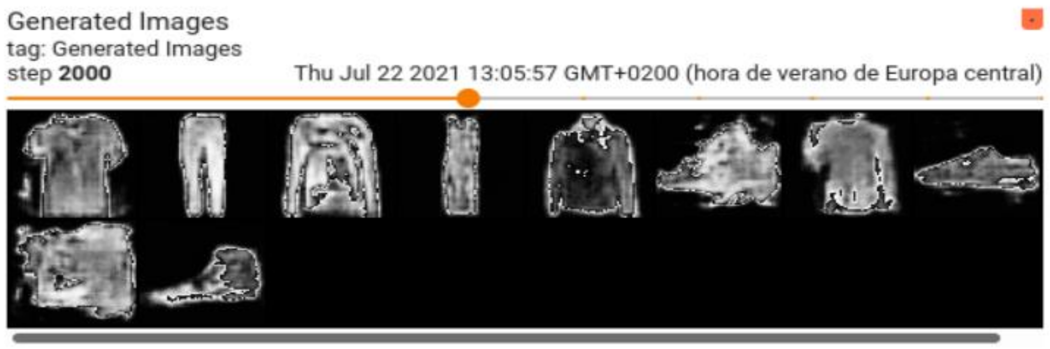

Training images
tag: Training images
step **0**
Thu Jul 22 2021 12:57:27 GMT+0200 (hora de verano de Europa central)

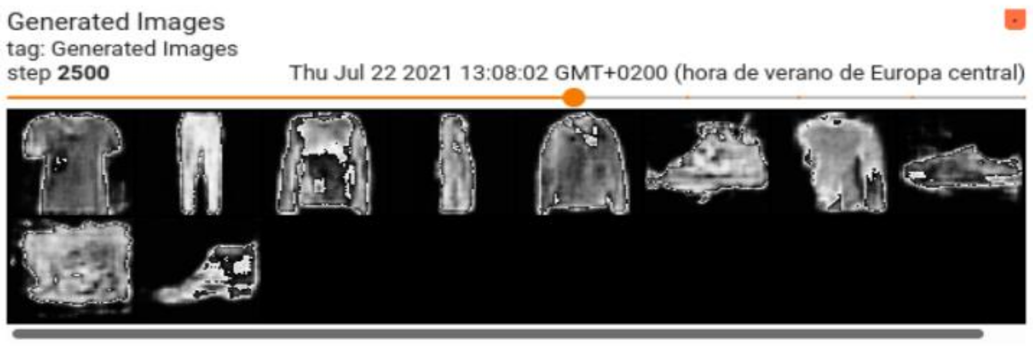

Generated Images
tag: Generated Images
step **0**
Thu Jul 22 2021 12:57:32 GMT+0200 (hora de verano de Europa central)

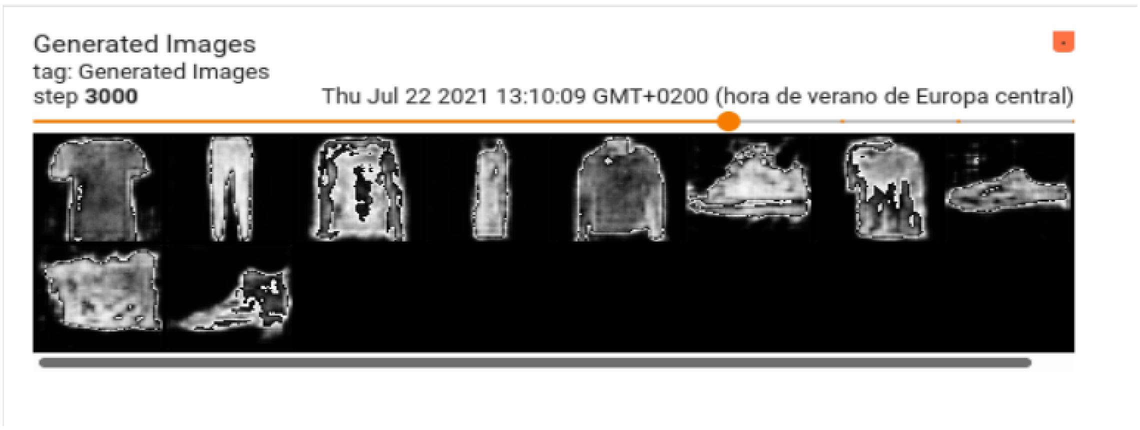

Generated Images
tag: Generated Images
step **500**
Thu Jul 22 2021 12:59:38 GMT+0200 (hora de verano de Europa central)

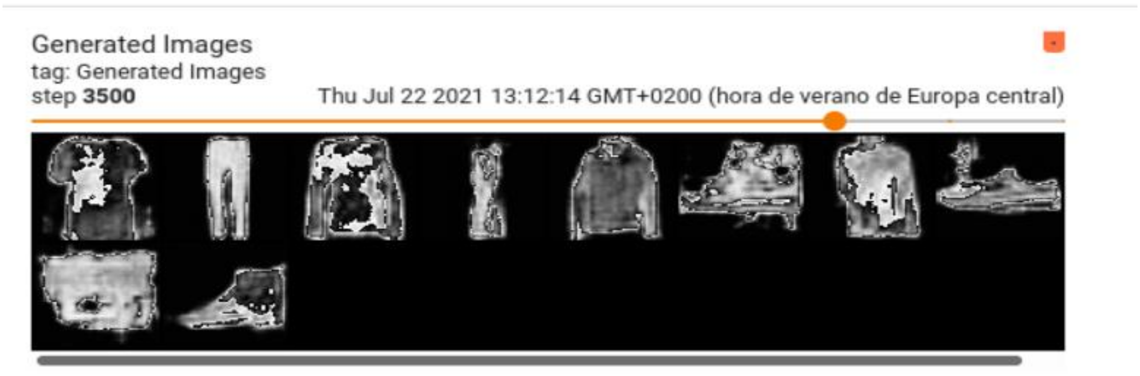

Generated Images
tag: Generated Images
step **1000**
Thu Jul 22 2021 13:01:45 GMT+0200 (hora de verano de Europa central)

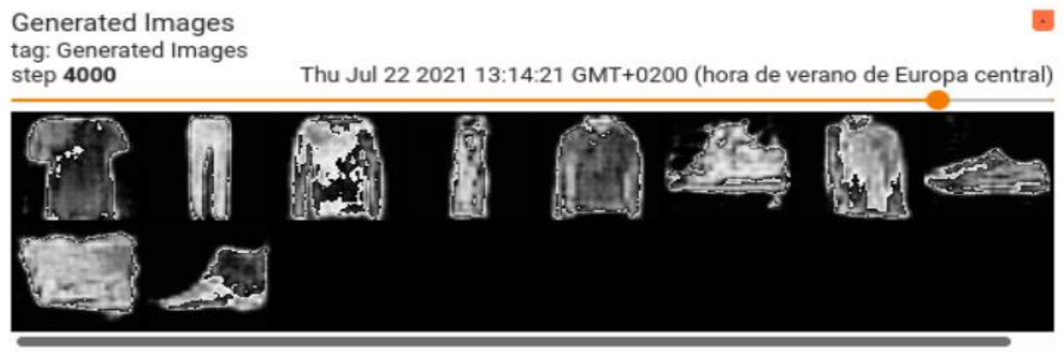

Generated Images
tag: Generated Images
step **1500**
Thu Jul 22 2021 13:03:50 GMT+0200 (hora de verano de Europa central)

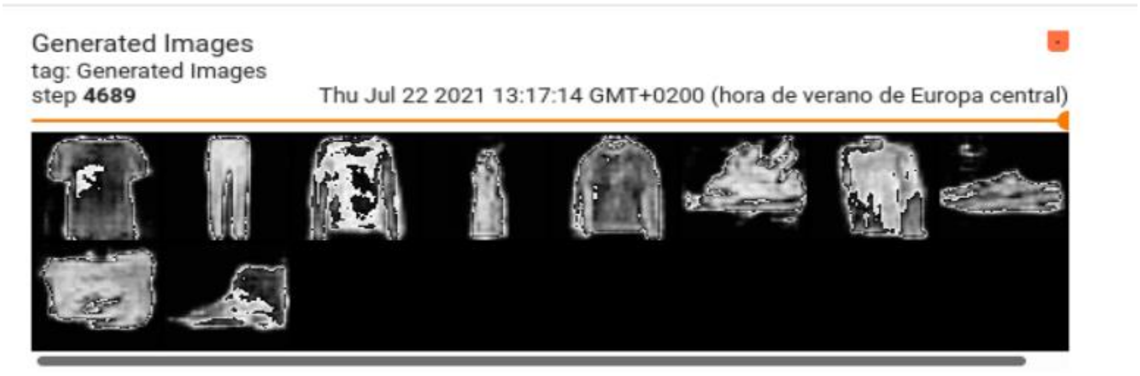

Generated Images
tag: Generated Images
step **2000**
Thu Jul 22 2021 13:05:57 GMT+0200 (hora de verano de Europa central)


Generated Images
tag: Generated Images
step **2500**
Thu Jul 22 2021 13:08:02 GMT+0200 (hora de verano de Europa central)


Generated Images
tag: Generated Images
step **3000**
Thu Jul 22 2021 13:10:09 GMT+0200 (hora de verano de Europa central)


Generated Images
tag: Generated Images
step **3500**
Thu Jul 22 2021 13:12:14 GMT+0200 (hora de verano de Europa central)


Generated Images
tag: Generated Images
step **4000**
Thu Jul 22 2021 13:14:21 GMT+0200 (hora de verano de Europa central)


Generated Images
tag: Generated Images
step **4689**
Thu Jul 22 2021 13:17:14 GMT+0200 (hora de verano de Europa central)