Chapter – 3 (implementation – Part 1 - Generator)

**Computer Vision**

**GANs: Image Generation**

Import Libraries

Generator Definition

Forward function for Generator

Generator Object

**3.1. Description & Import Libraries**

One of the application of GANs is to generate some fake images based on the vision of real images and we're gonna do that in this chapter.

* GANs will have eyes through the convolutional layers to watch all these images and then train itself to reproduce some fake images.
* Other amazing applications of GANs e.g. style transfer or face warp (with cycle GANs). GANs are the cutting edge model in Computer Vision.
* In this time we will implement the whole model from scratch and we're not gonna use the pre-trained models as before.
* CIFAR-10: We're going to make a NN-architecture to create a brain which will have some kinda eyes to see through Convolutional Layers. Which will be capable of generating some fake images based on the vision of a lot and lots of real images.
* The data-set, the real images we'll get from CIFAR-10 which is a very well known data-set containing lots and lots of images on which our Deep Convolutional Gannes will be trained.
* Code Structure: We divided the whole code into 3 sections.

1. Generator: This time we're going to have two neural networks. We're going to have the neural network of the generator and the neural network of the discriminator.

* We start by importing all the libraries.
  + We're going to use torch vision to visualize the images.
* Hyper parameters: Then we set the values of some hyper parameters like the ***batch size***, the ***image size*** that is. We set the size of the generated images to 64 by 64.
* Transformations: Then we create some transformations to make sure that the input images compatible with the neural network of the generator.
* Load the data-set: Then we load the data set from the "data" folder that contains the **CIFAR-10** data set in batches.
  + We'll use **torch.utils.data.DataLoader** to get the images of the data set batch by batch.
  + We'll specify the size of the batch.
* We'll shuffle the data to get the images in a random order.
* We'll use two parallel threads that will load the data faster.
* Then we define the **weights\_init()** function that will initialize all the weights of the neural network. We'll apply this function to both the neural networks: Generator & Discriminator to initialize all the weights in the right way for the Adversarial Networks.
* Define GENERATOR's structure: We will define the architecture of the Generator NN, we name it G. It is a class.
* We'll define the forward function that will propagate the signal inside this neural network.
* Then we create the generator itself which will be an object of the G class.

1. Discriminator: We'll define the architecture of the discriminator. We'll create a class D which will contain the architecture itself.

* We'll also define a forward function that will propagate the signal inside the neural network of the discriminator.
* Then we'll create an object of this D class.

1. Train: We will train the DC-GANs by training two brains GENERATOR & DISCRIMINATOR at the same time.

* We first train the GENERATOR with a real image of the data set.
* Then we train the DISCRIMINATOR with a fake image generated before by the GENERATOR.
* Finally the folder named "result" will be populated with the fake images of our decisions.

# *---------------    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)

**3.2. Generator Definition**

* We'll define the generator in two steps:
* First we'll make/create the class that will define the architecture of the neural network of the generator.
* We will put in a sequence to different modules that is the different layers of the neural network.
* Then we'll make the forward function to propagate the signal inside the NN.
* Once this class is done we'll be able to create the generator by creating an object (instance) of this class.
* We're going to use inheritance. We are going to inherit from **nn.module** (which contains all the tools that allow us to build a neural network) and it's called **nn.module** because the modules are the different applications and connections you can make inside a neural network.
* For example one ***module*** can be a ***convolution***.
* Another ***module*** can be a **full connection**.

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

* Now inside this new class we have to define the whole architecture. And besides we'll include the forward function to be able to afford propagate a signal.

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

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

* Notice **super**(G, self), we're using nn.Module's **\_\_init\_\_**() not G-class's . We used it to activate the inheritance.
* Our object is nothing else than the NN (the Generator) and the Generator will be composed of the different modules e.g the convolutions the full-connections and more.
* Meta module: Now we need to create a **meta module**. A meta module is a huge module that will be *composed of several* modules.
* By module we mean the *different layers* & *different connections* inside the Neural Network.
* Our *meta module* will be a sequence of several **Convolutions**, **Full-Connections**, **Inversed-Convolutions** and more.
* self.main: This will be our meta module. This **self.main** will contain all the different modules in a sequence of layers.
* Note that, **self.main** is an object of **nn.sequential** class.
* Also note that we apply all other modules as several parameters of this meta module.

**Architecture of the NN (Generator):**

The architecture is coming from a lot of experimenting that was done by researchers, the machine learning and AI contributors. It is open source and that turns out to work very well for Adversarial Networks. So don't worry about the choice of the numbers. They just come from experimentation.

* Inverse Convolution: Recall a CNN takes as input (some images) and returns as output a vector.
* But the inverse CNN (Inverse Convolution NN) will do exactly the opposite. Since the generator will generate some fake images.
* It will take a vector as inputs (we'll call it **noise**, a **random vector** of **size 100**) and it will return a fake image.

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

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

        # *meta module of a NN 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.*

        )

* To get the inverse of a convolution in PyTorch we're going to use **nn.ConvTranspose2d** class.

**nn.ConvTranspose2d**(100, 512, 4, 1, 0, bias = **False**),

* Arguments:
* Size of the INPUT: 100, meaning that the inputs of inverse CNN will be a vector of ***size 100***.
* Number of feature maps of the OUTPUT: 512
* Size of the KERNEL: **4**, which means that the **kernels** will be squares of size **4 by 4**.
* STRIDE: **1**
* PADDING: **0**.
* BIAS: By default **True**. So we set it **False**.
* Batch-Normalization: Now we are going to normalize all the features along the dimension of the Batch. Since the diamonds and of the batch is **512**. We have **512 feature maps** and we're going to apply **BatchNorm** on each of these **512 feature maps**.

**nn.BatchNorm2d**(512),

* We used 512 as argument since we have 512 features.
* RELU: Then we are going to apply RELU-rectification for the non-linearity of the neural network that is to break the linearity.

**nn.ReLU**(**True**),

* Repeat the process: Now we are going to repeat the above process (Inverse-Convolution, Batch-Normalization & RELU) but we'll change the parameters each time.

**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.*

* Notice the new parameters of **ConvTranspose2d,** since the output-size of previous layer is **512** (Number of feature maps of the OUTPUT) we used **512** here as **input-size**, the new **output-size** is set to **256** (it's the half of the input size), kernel-size is the same, now the **stride** and **padding** are **2** & **1** respectively.
* We apply **BatchNorm2d** for our new **256** feature-map output.
* We apply the ReLU again

**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.*

* In this time we only change the **Input** & **Output** Size to **256** & **128** respectively. We apply **BatchNorm2d** for our new **128** feature-map output and apply the ReLU again

**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.*

* Again we change the **Input** & **Output** Size to **128** & **64** respectively. We apply **BatchNorm2d** for our new **64** feature-map output and apply the ReLU again.
* Notice the **input** and **output** is reduced to **half** for each module.
* The last inverse-CNN module: This time we are going to input different arguments.

**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.*

* The first argument is the number of inputs which is the output of the **previous inverse convolution**. So that's 64 again.
* Since we're making the generator, which generates some fake images, and these fake images are going to be with three channels. Hence, the output of the generator is exactly going to be the three channels of the fake images.
* Therefore the number of dimensions for the output is going to be 3.
* Other parameters are the same.
* At the end, for the *rectification*, we use Hyperbolic Tangent Rectification to not only break the linearity again but also to make sure we are between **minus 1** and **plus 1** and also it is **centered** around **0**.
* Why do we want to output values to fall between -1 and +1 and being centered around 0: That's because we want the same standards as the images of the data set. So that the generated images from the generator will become the input of the **discriminator**.

Next we'll make the forward function that will propagate the signal (i.e. the input of the generator) through all the different layers of this newly created NN.

**3.3. Forward function for Generator**

Now we'll define the forward function inside our Generator definition that will forward propagate the signal inside the whole Neural Network of the Generator.

* We will simply use the **main** object (object of the sequential class, which is a meta-module) that contains all the different modules to forward propagate the signal.
* **forward()** takes input as an argument that will be **fed** to the **NN**, and that will return the output containing the generated images.
* We forward propagate the signal *through the whole neural network* of the *generator* defined by **self.main**.

**def** **forward**(self, input):

        output = self**.main**(**input**)

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

* Arguments:
* **self** represents the neural network itself (NN of the generator). We'll use this to apply **main** module.
* **input** is the input of the NN of the generator.
* I remind this input is going to be some random vector of size 100. Which we defined in our first inverse convolution.
* This is a random vector input of the generator which will just represent some noise to generate a fake image. It will be the (output) fake image generated by the generator.
* We'll create this random vector noise that will be exactly this input.
* It'll be useful when we start to train the brain of the generator and the brain of the discriminator.
* Next we'll create an instance this Generator class G.
* The object will be the neural network of the generator that we defined above.
* We'll use **apply**(weights\_init) to initialize the weights of the NN as it should be. That is according to the conventions of the adversarial networks.

**3.4. Generator Object**

Previously we have defined the Generator G class containing the architecture and forward function. Now we are ready to create as many objects as we want i.e. as many generators as we want. Now we only need one object of the Generator.

# *Creating the generator*

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

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

* There is no arguments in netG = **G**(), because in the definition of **G**, we just used inheritance and not any real arguments. So basically there is no argument.
* **weights\_init**: We need to initialize the weights, the proper way to respect the *Convention* of the *Adversarial Networks*. To do this we have the **weights\_init** function.

# *Defining the weights\_init function that takes as input a NN 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)

* Notice the **classname** variable, it'll look for some names in the definition of the **NN-class m** (in our case G). It'll check the names **'Conv'** and **'BatchNorm'**.
* We've used **ConvTranspose2d** in the definition of the generator **G**, and **ConvTranspose2d** contains 'Conv'.
* So **weights\_init** will find **ConvTranspose2d** and then it will initialize the weights to (0.0, 0.02) for the convolution modules.
* Same for the 'BatchNorm', we've also used **BatchNorm2d** in the definition of **G**. **weights\_init** going to look for any name in the class that contains 'BatchNorm'which of course the **BatchNorm2d** to batch normalize the feature map.
* On each of the layers related to the 'BatchNorm' will be initialized to the weights to (1.0, 0.02).
* And remember in each layer we also have some **bias** and all the **bias** at the batchnorm levels will be **initialized** to **zero**.
* To apply this **weights\_init()** function we just need to take our *object of generator neural network* **netG**, and we use **apply()** on it.

**netG.apply**(weights\_init)

Now have a real generator neural network. Next, we'll define and create the discriminator.

**All codes at once with Generator Definition**

# *---------------    DC 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.*