Chapter – 3 (implementation – Part 2 - Discriminator)

**Computer Vision**

**GANs: Image Generation**

Discriminator Definition

Forward function for Discriminator

Discriminator Object

**3.5. Discriminator Definition**

Previously we created architecture of the generator. Now we create the second brain, the brain of the DISCRIMINATOR of our Deep Convolutional GANs.

* First we define the class for the DISCRIMINATOR. Inside this class we create a meat-module as before and a forward function. After that we use this class to create a object, which will be the Discriminator NN for the GAN.
* We call the class of our Discriminator D which is going to inherit from the nn.Module.

**class** D(nn.Module):

* Activating the inheritance:

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

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

* Meta module: Before we used Inverse-Convolution for **Generator**, but for the **Discriminator** we'll use Convolution.

        # *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.*

        )

* As before, our meta module **self.main** is an object of **nn.sequence**.
* The *Discriminator* takes as *inputs* a *generated image* coming from the *generator* and return as *output* a *discriminating number* which will be of value between 0 and 1.
* Since ***Discriminator*** takes *image* as *inputs* and *outputs* a *value*, we're going to use convolution not an inverse convolution.
* Through the multiple convolution we will get a simple vector of one element that will contain the discriminating number (a value between 0 and 1).
* As before, we're going to add a *series of different convolutions* with *different parameters* and we'll **BatchNorm** the feature maps then we'll apply some rectification (this time we'll use **LeakyReLU** instead of **ReLU**).
* First Layer:

**nn.Conv2d**(3, 64, 4, 2, 1, bias = **False**)

* Since the input in now the generated image by the generator (output from the generator), it has 3 channels, we choose **3** for the input size, **64** for feature maps. **4** for the kernels, stride is **2**, padding is **1**, and no bias.
* Rectification: We're going to use a **LeakyReLU**. We have a slight change from **ReLU**, here **LeakyReLU** gives some negative values for the negative values of **x**.

|  |  |
| --- | --- |
| **LeakyReLU** | **ReLU** |
|  |  |

* Why do we choose to use a **LeakyReLU** instead of **ReLU**: It comes from experimentation and research, basically for the convolutions of the discriminator it works better with the **LeakyReLU**.

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

* Negative slope: we set 0.2 (it's by experimentation , we're not gonna choose 0.01).
* inplace: we set it to true to benefit from this option. It do the operations inplace (more details on PyTorch documentation).
* 2nd Layer of convolution: ***Input size*** is now ***64*** (output from ***previous layer***), ***output-size*** is ***128***, and other parameters remains same.
* Notice that we are doing convolutions so we are kind of doing reverse as we did for the Generator Network. We are increasing the feature-maps.
* To normalize the features, we use **BatchNorm2d()**, and we use **128**, since we have **128** **feature maps** now.
* Then we apply the 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.*

* 3rd and 4th layers: Similarly we implement third and forth layer of the Discriminator NN.

**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.*

* Last layer: The **input-size** is now **512**, for the **output size** we set **1**, to specify the **final output**.
* Since a discriminator returns a discriminating number between 0 and 1, so that's a simple vector of one dimension containing this number.
* We keep the **kernel size** of **4**, we change the **stride** to **1** and **padding** to **0** and **no bias**.
* For the generator we applied a hyperbolic tangent activation function.
* But this time for discriminator we'll apply a sigmoid function because it returns some values **between** **0** and **1** and for the discriminator we actually want it to be **between** **0** and **1**.
* It's for discriminating reason. The principle of the discriminator is that zero corresponds to the rejection of the image and one corresponds to the acceptance of the image.
* It'll use a classic threshold technique, for the values below 0.5 we will consider it as zeros & it will reject the image and for the values above 0.5 we will consider it as one and we will accept the image.
* So the sigmoid function that not only break down linearity but also returns a value between 0 and 1 is the optimal choice for the activation function at the level of the output of the discriminator.

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

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

Next, well build the forward function for the Discriminator NN.

**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.*

        )

**3.6. Forward function for Discriminator**

We're going to make a new forward function this time for the discriminator. It is mostly similar what we did for the generator.

* We will simply use the **main** object (the meta-module for the Discriminator) that contains all the different modules to forward propagate the signal through the Discriminator.
* **forward()** takes input as an argument that will be **fed** to the **NN**.
* We forward propagate the signal *through the whole neural network* of the *Discriminator* defined by **self.main**.

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

# *We forward propagate the signal through the whole neural network of the discriminator defined by self.main.*

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

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

* Arguments:
* **self** represents the neural network itself (NN of the *Discriminator*). We'll use this to apply **main** module.
* **input** is the input of the NN of the generator.
* Here the input is the generated image from the Generator. The input is of three dimensions, corresponding to the three channels.
* And keep in mind that the output of the discriminator and therefore of this forward function is a discriminating number between **0** and **1**. For a number close to **0**, the image will be rejected and for number close to **1** will accept the image.
* Return Output: To return the output, we use a trick. It's actually slightly technical.
* If we have a better look at the architecture of the neural network of the discriminator we see that it's actually a **sequence of convolutions**.
* It's a CNN, a CNN is composed of several convolutions, which we've done right here.
* At the ***end*** of the ***CNN*** we need to ***flatten*** the result of all the convolutions (i.e. the result after applying the last convolution).
* Flattening: Now we have to flatten the result of the convolutions so that all the elements of the output are along one same dimension.
* This dimension corresponds to the dimension of the batch size.
* In **PyTorch** to apply flattening we use **view(-1)**.

**return** **output.view**(-1)

* This means we want to **flatten** the result of the **convolutions** that at the end are in **2D**, into one same dimension along one same flattened vector.
* Let's recap the functionality:
* The discriminator takes as inputs an image created by the generator.
* Discriminator will decide if it wants to accepted or rejected the image. To make that decision it will output a discriminating number between **0** and **1**.
* For a number close to **0**, the image will be rejected and for number close to **1** will accept the image.
* Hence, in some way the discriminator is discriminating the creations of the generator.
* The discriminator is an adversary of the generator, judging the creations of the generator that the image should be accepted or not. That's why the name "**Generative Adversarial Networks**" is chosen.

Next we'll make an object of this discriminator class to make the brain of the discriminator.

**3.7. Discriminator Object**

We'll create an object of the discriminator class to make the brain of the discriminator. It's exactly the same as what we did for the brain of the generator.

# *Creating the discriminator*

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

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

* We create an object of the discriminator class **D**.
* Then we initialize the weight using **apply(weights\_init)**.
* Next we'll move to the training phase. We'll train these two brains according to the process of the generative adversarial networks and let's give
* It has two big steps:

1. The first step is to ***update*** the ***weight*** of the ***discriminator*** neural network and that consists of ***maximizing*** the ***error*** calculated by the ***BCE-loss***.
2. Second step is to ***update*** the ***weights*** of the ***generator*** which again will consist of ***maximizing*** the ***error*** calculated by the ***BCE-loss***.

**All code at once (Discriminator)**

# *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 takes an input that will be fed to the NN, and return the output between 0 and 1.*

        output = self**.main**(**input**) # *forward propagate the signal through the whole NN 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.*