

12_GANs_final

February 5, 2020

1 Assignment 12

In this tutorial, we will work on GANs networks (<https://arxiv.org/pdf/1406.2661.pdf>). This kind of network is part of the generative networks who are actually able to generate synthetic data. This network is actually composed of two, a generator responsible for generate fake data and a discriminator for investigating on the trueness of the images.

```
In [2]: from IPython.display import Image
        from IPython.core.display import HTML
        Image(url= "https://pathmind.com/images/wiki/GANs.png", width=700)
```

```
Out[2]: <IPython.core.display.Image object>
```

The discriminator can be see as a very simple fully connected network, composed of several layer of neurons and a simple output of one neurone who just determine if a image is true or false.

The generator looks more like the upscaling part of the segmenetation network. A small piece of noise is injected and the output is an images.

The training is maybe the most original part.

First, you train the discriminator alone by injecting the training set (True data so label=True) and a set of data generated by the generator (label=False) you then have a loss to back propagate. *Applause, you trained you discriminator for one batch*

Second, you inject some noise in your generator, the generator produce some fake images, you inject this fake images in you discriminator and use the labels generated by the discriminator to compute your backpropagation. So to train your generator, you will consider the generator and the discriminator as the same network, you just don't train the discriminator in this case.

1.1 12.1

We ask you then to program you own GANs. this network should be just compose of fully connected layer (no conv). You should define the architecture of the network, then provide two training function one for the generator and one for the discriminator and make him learn!

We will take the MNIST dataset here for simplicity and lightness. The architecture I used was :
G : (Input : random Tensor = 100) => (FC1 : fully connected = 256) => (BatchNormalisation) => (LeakyReLU) => (DropOut) => (FC2 : fully connected = 512) => (BatchNormalisation) => (LeakyReLU) => (DropOut) => (FC2 : fully connected = 784) => (Tanh)

D : (Input : flatten images Tensor = 784) => (FC1 : fully connected = 1024) => (BatchNormalisation) => (LeakyReLU) => (DropOut) => (FC2 : fully connected = 512) => (BatchNormalisation) => (LeakyReLU) => (DropOut) => (FC2 : fully connected = 1) => (Sigmoid)

This architecture is what I use, it's working but not great, so feel free to change it!!

The training is 20-30min on a 1060 6Go for 100 epochs and batch size of 100. Here, it's a little bit different than before, the loss do not necessarily go down, it's a race between D and G so both cannot be good at the same time.

```
In [0]: # Hyper parameters
GENERATOR_INPUT_SIZE = 100
BATCH_SIZE = 100
LEARNING_RATE_G = 0.01
LEARNING_RATE_D = 0.01
MOMENTUM_G = 0.5
MOMENTUM_D = 0.5
NUM_EPOCHS = 25
K = 4
# batch_size_train = 10
# batch_size_test = 1000
# log_interval = 10

# random_seed = 1
# torch.backends.cudnn.enabled = False
# torch.manual_seed(random_seed)

In [4]: import os
import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F

from torchvision import transforms
import torch.optim as optim
from torchvision.utils import save_image
from torch.autograd import Variable
import matplotlib.pyplot as plt
import pylab
import numpy as np

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Image processing
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize(mean=(0.5,), std=(0.5,))])

# MNIST dataset
train_mnist = torchvision.datasets.MNIST(root='./data/',
                                          train=True,
                                          transform=transform,
                                          download=True)
```

```
# Data loader
```

```
train_loader = torch.utils.data.DataLoader(dataset=train_mnist,  
                                             batch_size=BATCH_SIZE,  
                                             shuffle=True)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./data/MNIST/raw/train-images-idx3-ubyte.gz
```

```
HBox(children=(IntProgress(value=1, bar_style='info', max=1), HTML(value='')))
```

```
Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ./data/MNIST/raw/train-labels-idx1-ubyte.gz
```

```
HBox(children=(IntProgress(value=1, bar_style='info', max=1), HTML(value='')))
```

```
Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw
```

```
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw/t10k-images-idx3-ubyte.gz
```

```
HBox(children=(IntProgress(value=1, bar_style='info', max=1), HTML(value='')))
```

```
Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw
```

```
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz
```

```
HBox(children=(IntProgress(value=1, bar_style='info', max=1), HTML(value='')))
```

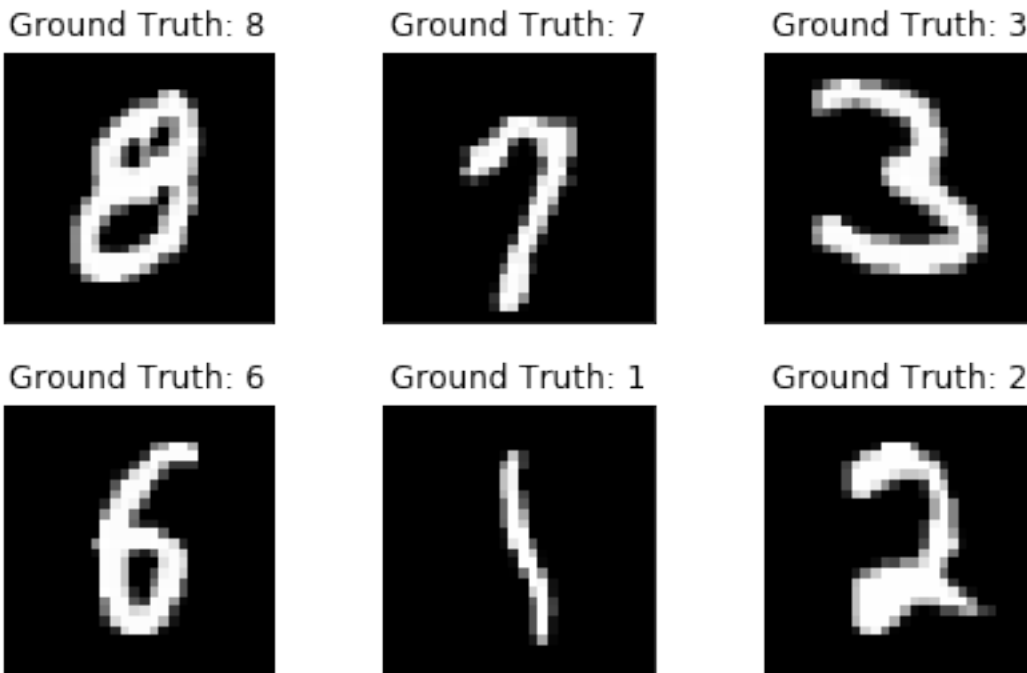
```
Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw
```

```
Processing...
```

```
Done!
```

```
In [5]: examples = enumerate(train_loader)  
        batch_idx, (example_data, example_targets) = next(examples)  
        example_data.shape  
  
        fig = plt.figure()  
        for i in range(6):  
            plt.subplot(2, 3, i + 1)
```

```
plt.tight_layout()
plt.imshow(example_data[i][0], cmap='gray', interpolation='none')
plt.title("Ground Truth: {}".format(example_targets[i]))
plt.xticks([])
plt.yticks([])
```



```
In [0]: class Discriminator(nn.Module):
        """
        D :
        (Input : flatten images Tensor = 784)
        => (FC1 : fully connected = 1024)
        => (BatchNormalisation)
        => (LeakyReLU)
        => (DropOut)
        => (FC2 : fully connected = 512)
        => (BatchNormalisation)
        => (LeakyReLU)
        => (DropOut)
        => (FC3 : fully connected = 1)
        => (Sigmoid)
        """

        def __init__(self):
            super().__init__()
```

```

self.seq = nn.Sequential(
    nn.Linear(28**2, 1024),
    nn.BatchNorm1d(1024),
    nn.LeakyReLU(),
    # Not more than 25%, http://iamtrask.github.io/2015/07/28/dropout/
    nn.Dropout(p=0.2),

    nn.Linear(1024, 512),
    nn.BatchNorm1d(512),
    nn.LeakyReLU(),
    nn.Dropout(p=0.5),

    nn.Linear(512, 1),
    nn.Sigmoid(),
)

def forward(self, img):
    # print(img.shape)
    flattened_img = img.view(img.size(0), -1)
    return self.seq(flattened_img)

class Generator(nn.Module):
    """
    G :
    (Input : random Tensor = 100)
    => (FC1 : fully connected = 256)
    => (BatchNormalisation)
    => (LeakyReLU)
    => (DropOut)
    => (FC2 : fully connected = 512)
    => (BatchNormalisation)
    => (LeakyReLU)
    => (DropOut)
    => (FC2 : fully connected = 784)
    => (Tanh)
    """

    def __init__(self):
        super().__init__()
        self.seq = nn.Sequential(
            nn.Linear(GENERATOR_INPUT_SIZE, 256),
            nn.BatchNorm1d(256),
            nn.LeakyReLU(),
            # Not more than 25%, http://iamtrask.github.io/2015/07/28/dropout/
            nn.Dropout(p=0.2),

            nn.Linear(256, 512),

```

```

        nn.BatchNorm1d(512),
        nn.LeakyReLU(),
        nn.Dropout(p=0.5),

        nn.Linear(512, 784),
        nn.Tanh(),
    )

    def forward(self, z):
        x = self.seq(z)
        img = x.view(x.size(0), 1, 28, 28)
        return img

```

```

In [0]: # Loss function:
        # In the paper, the GAN corresponds to a minimax 2-player game.
        # As of
        # https://developers.google.com/machine-learning/gan/loss#minimax-loss
        # "The formula derives from the cross-entropy between the real and generated distributions."
        # Thus, we use a cross-entropy loss function pytorch provides:
        # 'CrossEntropyLoss' resulted in an error so we tried 'BCELoss'. ;)
        criterion = nn.BCELoss().to(device)
        # criterion = nn.CrossEntropyLoss()

        G = Generator().to(device)
        D = Discriminator().to(device)

        optimizer_G = optim.SGD(G.parameters(), lr=LEARNING_RATE_G, momentum=MOMENTUM_G)
        optimizer_D = optim.SGD(D.parameters(), lr=LEARNING_RATE_D, momentum=MOMENTUM_D)

```

```

In [0]: def train(epochs=NUM_EPOCHS, k=K):
        """
        Algorithm 1:
        Minibatch stochastic gradient descent training of generative adversarial nets.
        The number of steps to apply to the discriminator, k, is a hyperparameter.
        We used k = 1, the least expensive option, in our experiments.

        for number of training iterations do
            for k steps do
                Sample minibatch of m noise samples {z(1), ... , z(m)} from noise prior pg(z)
                Sample minibatch of m examples {x(1), ... , x(m)} from data generating distribution
                Update the discriminator by ascending its stochastic gradient
            end for
            Sample minibatch of m noise samples {z(1), ... , z(m)} from noise prior pg(z)
            Update the generator by descending its stochastic gradient
        end for

        The gradient-based updates can use any standard gradient-based learning rule.
        We used momentum in our experiments.

```

=====

The training is maybe the most original part:

First, you train the discriminator alone by injecting the training set (True data so label=True) and a set of data generated by the generator (label=False) you then have a loss to backpropagate. Applause, you trained your discriminator for one batch.

Second, you inject some noise in your generator, the generator produce some fake images. You inject these fake images into your discriminator and use the labels generated by the discriminator to compute your backpropagation.

So to train your generator, you will consider the generator and the discriminator as the same network, you just don't train the discriminator in this step.

```
for epoch in range(epochs):
    for batch_idx, data in enumerate(train_loader):
        images, _labels = data
        images = images.to(device)
        batch_size = images.shape[0]
        labels_real = torch.ones(batch_size, 1,
                                device=device,
                                requires_grad=False)
        labels_fake = torch.zeros(batch_size, 1,
                                device=device,
                                requires_grad=False)

        # DISCRIMINATOR
        for step in range(k):
            optimizer_D.zero_grad()
            # > First, you train the discriminator alone
            # > by injecting the training set (true data so label=True)
            output_real = D(images)
            # > and a set of data generated by the generator (label=False).
            # Create random/noise tensors (z space) as inputs for the generator.
            z = torch.randn(batch_size, GENERATOR_INPUT_SIZE)
            generated_imgs = G(z.to(device))
            # '.detach' is necessary because we're using the same variable again later
            # but the graph is already freed at that point.
            output_fake = D(generated_imgs.detach())
            # > You then have a loss to back propagate.
            real_loss = criterion(output_real, labels_real)
            fake_loss = criterion(output_fake, labels_fake)
            # loss_D = (real_loss + fake_loss) / 2
            loss_D = real_loss + fake_loss
            loss_D.backward()
            optimizer_D.step()
            # > Applause, you trained your discriminator for one batch.

        # GENERATOR
```

```

optimizer_G.zero_grad()
# > Second, you inject some noise in your generator,
# > the generator produces some fake images.
# > You inject these fake images into your discriminator
output_D = D(generated_imgs)
# > and use the labels generated by the discriminator
# > to compute your backpropagation.
loss_G = criterion(output_D, labels_real)
loss_G.backward()
optimizer_G.step()
# > So to train your generator, you will consider the generator and
# > the discriminator as the same network,
# > you just don't train the discriminator in this case.


#         images, _labels = data
#         images = images.to(device)
#         batch_size = images.shape[0]
#         labels_real = torch.ones(batch_size, 1,
#                                   device=device,
#                                   requires_grad=False)
#         labels_fake = torch.zeros(batch_size, 1,
#                                    device=device,
#                                    requires_grad=False)

#         optimizer_D.zero_grad()
#         optimizer_G.zero_grad()

#         # > First, you train the discriminator alone
#         # > by injecting the training set (true data so label=True)
#         output_real = D(images)
#         # > and a set of data generated by the generator (label=False).
#         # Create random/noise tensors (z space) as inputs for the generator.
#         z = torch.randn(batch_size, GENERATOR_INPUT_SIZE)
#         generated_imgs = G(z.to(device))
#         # '.detach' is necessary because we're using the same variable again l
#         # but the graph is already freed at that point.
#         output_fake = D(generated_imgs.detach())
#         # > You then have a loss to back propagate.
#         real_loss = criterion(output_real, labels_real)
#         fake_loss = criterion(output_fake, labels_fake)
#         # loss_D = (real_loss + fake_loss) / 2
#         loss_D = real_loss + fake_loss
#         loss_D.backward()
#         # > Applause, you trained you discriminator for one batch.


#         # > Second, you inject some noise in your generator,

```



```

#             # > the generator produces some fake images.
#             # > You inject these fake images into your discriminator
#             output_D = D(generated_imgs)
#             # > and use the labels generated by the discriminator
#             # > to compute your backpropagation.
#             loss_G = criterion(output_D, labels_real)
#             loss_G.backward()
#             # > So to train your generator, you will consider the generator and
#             # > the discriminator as the same network,
#             # > you just don't train the discriminator in this case.

#             optimizer_G.step()
# #             optimizer_D.step()

print(
    "[epoch %d/%d] [batch %d/%d] [loss D: %f] [loss G: %f]"
    % (
        epoch, epochs,
        batch_idx, len(train_loader),
        loss_D.item(),
        loss_G.item(),
    )
)

```

In [0]: train()

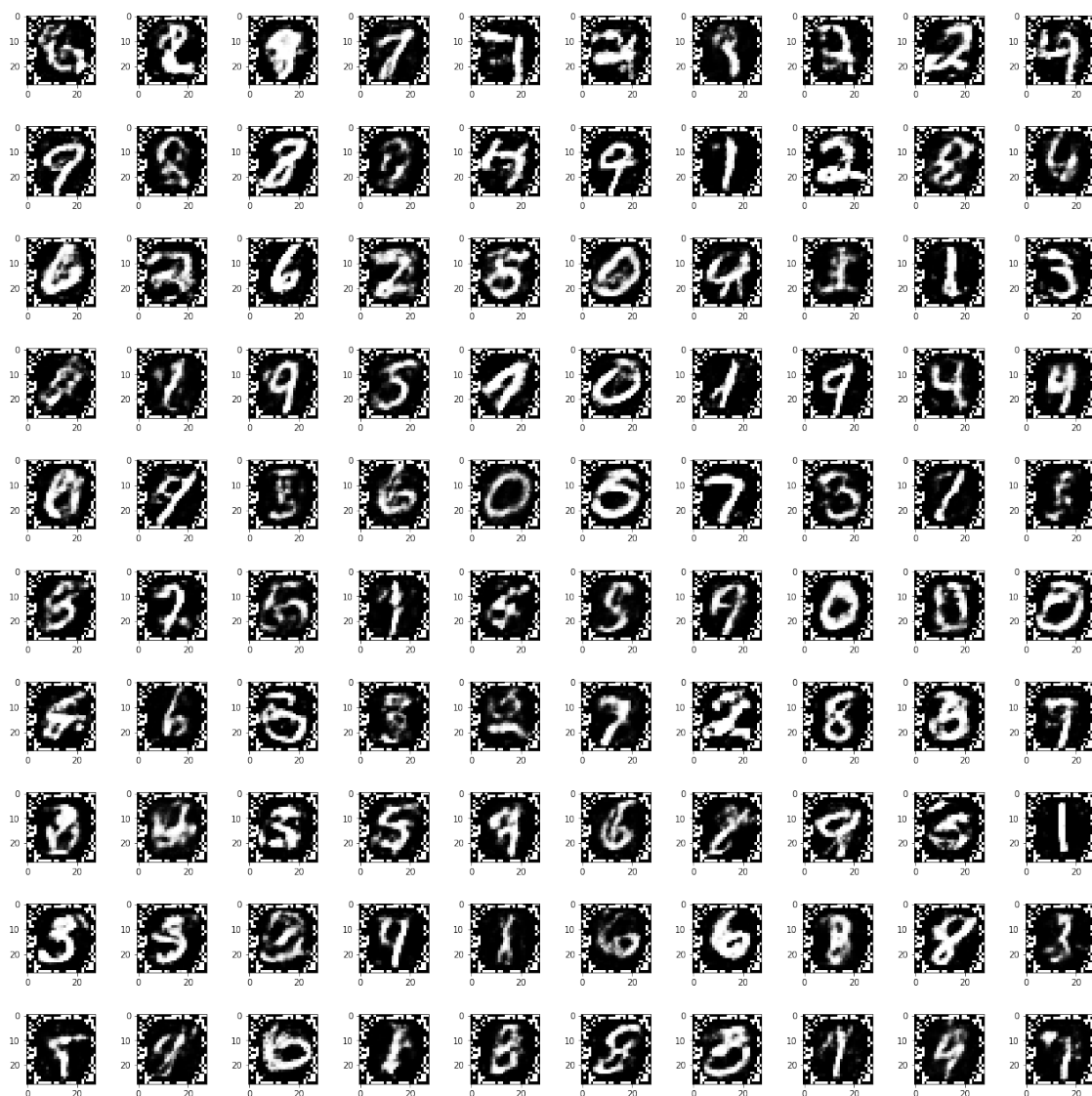
```

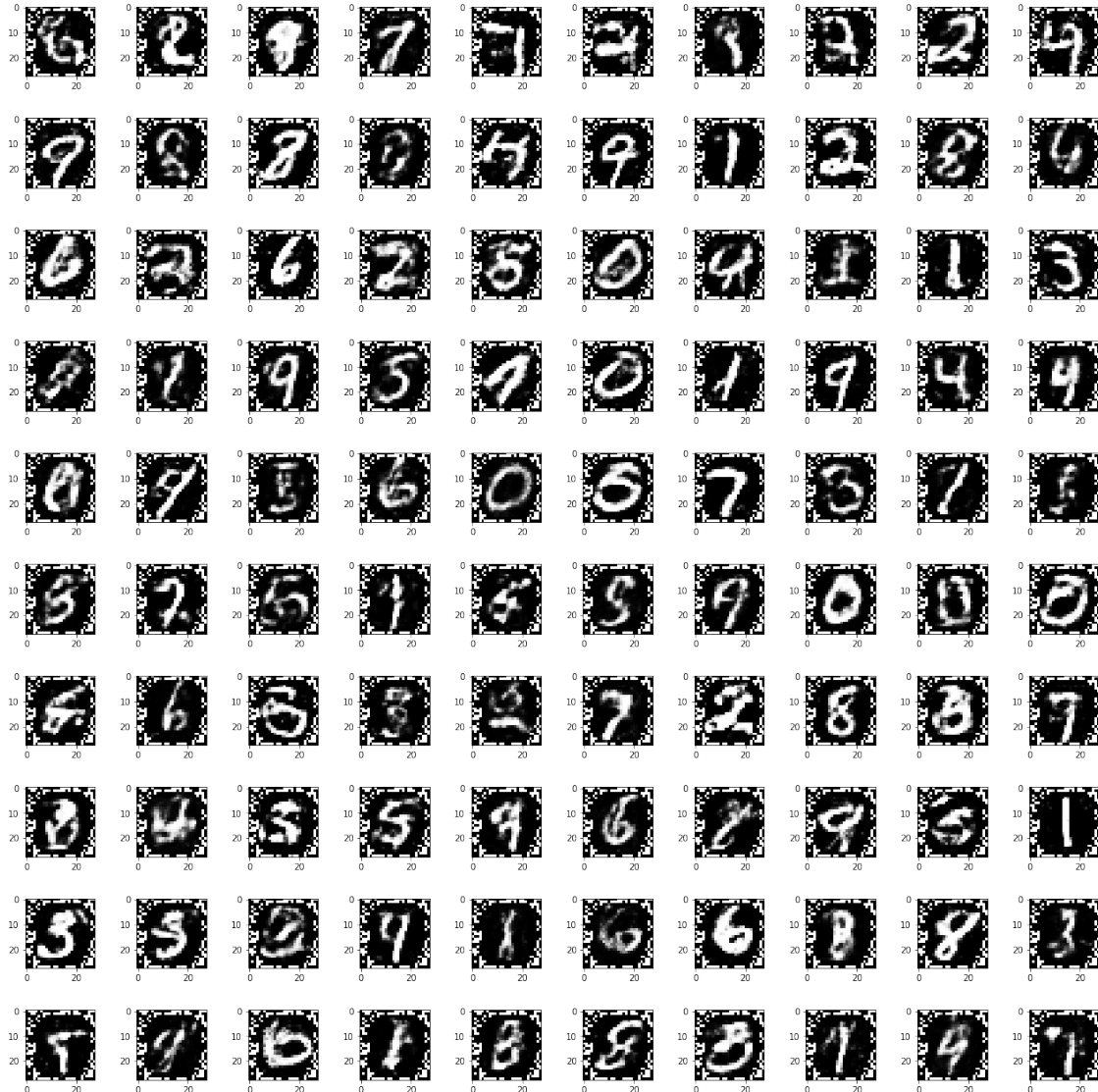
In [9]: z1 = torch.randn(BATCH_SIZE, GENERATOR_INPUT_SIZE).to(device)
        gen_imgs = G(z1).detach().cpu()

        fig, axes = plt.subplots(nrows=10, ncols=10)
        fig.set_figwidth(18)
        fig.set_figheight(18)
        fig.tight_layout()
        ax = np.asarray(axes).ravel()
        for i in range(len(gen_imgs)):
            ax[i].imshow(gen_imgs[i][0], cmap='gray', interpolation='none')
        fig

```

Out [9]:





1.2 8.2 Bonus

For the bonus, fancy stuff we would like you to program a conditional GAN. This GAN should produce the output you want. Instead of feeding your generator with just noise, you feed it with noise and label and it should generate the number you want. To do so, you add a label part to your Generator's input and you tell your discriminator what number it should recognize by adding a label part to your Discriminator's input.

```
In [9]: Image(url= "https://raw.githubusercontent.com/znxlwm/pytorch-MNIST-CelebA-cGAN-cDCGAN/main")
```

```
Out[9]: <IPython.core.display.Image object>
```

```
In [10]: # One-hot
def onehot(label):
```

```

        z = torch.zeros((100,10))
        z[:,label] = 1
        return z

    print(onehot(7).shape)
    #print(onehot(7))

torch.Size([100, 10])

In [0]: # Generator and Discriminator cGAN
class cGenerator(nn.Module):
    def __init__(self, label):
        super(cGenerator, self).__init__()
        # onehot - y in the picture
        self.oh = onehot(label).to(device)
        self.y = nn.Linear(10, 256)
        # noise z
        self.z = nn.Linear(GENERATOR_INPUT_SIZE, 256)

        self.bn256 = nn.BatchNorm1d(256)
        self.relu = nn.ReLU()

        self.bn512 = nn.BatchNorm1d(512)

        self.fc0 = nn.Linear(512, 512)

        self.fc1 = nn.Linear(512, 1024)
        self.bn1024 = nn.BatchNorm1d(1024)

        self.fc2 = nn.Linear(1024, 784)
        self.tanh = nn.Tanh()

    def forward(self, noise):
        y = self.relu(self.bn256(self.y(self.oh)))
        z = self.relu(self.bn256(self.z(noise)))

        x = torch.cat((z, y), dim=1)
        x = self.relu(self.bn512(self.fc0(x)))
        x = self.relu(self.bn1024(self.fc1(x)))

        return self.tanh(self.fc2(x))

class cDiscriminator(nn.Module):
    def __init__(self, label):
        super(cDiscriminator, self).__init__()
        # onehot - y in the picture
        self.oh = onehot(label).to(device)

```

```

self.y = nn.Linear(10, 1024)
# noise z
self.x = nn.Linear(28*28, 1024)

self.leaky = nn.LeakyReLU(0.2)
self.relu = nn.ReLU()

self.fc0 = nn.Linear(2048, 512)
self.bn512 = nn.BatchNorm1d(512)

self.fc1 = nn.Linear(512, 256)
self.bn256 = nn.BatchNorm1d(256)

self.fc2 = nn.Linear(256, 1)
self.sigmoid = nn.Sigmoid()

def forward(self, img):
    flattened_img = img.view(img.size(0), -1)

    y = self.leaky(self.y(self.oh))
    x = self.leaky(self.x(flattened_img))
    x = torch.cat((x, y), dim=1)
    x = self.leaky(self.bn512(self.fc0(x)))
    x = self.leaky(self.bn256(self.fc1(x)))

    return self.sigmoid(self.fc2(x))

```

```

In [0]: criterion = nn.BCELoss().to(device)
        # criterion_c = nn.CrossEntropyLoss()

G = cGenerator(0).to(device)
D = cDiscriminator(0).to(device)

optimizer_G = optim.SGD(G.parameters(), lr=LEARNING_RATE_G, momentum=MOMENTUM_G)
optimizer_D = optim.SGD(D.parameters(), lr=LEARNING_RATE_D, momentum=MOMENTUM_D)

In [13]: z_before_train = torch.randn(BATCH_SIZE, GENERATOR_INPUT_SIZE).to(device)
         gen_imgs = G(z_before_train).detach().cpu()
         gen_imgs_np = gen_imgs.numpy().reshape(100,28,28)

         print(gen_imgs_np.shape)
         print(type(gen_imgs_np))

fig, axes = plt.subplots(nrows=10, ncols=10)
fig.set_figwidth(18)

```

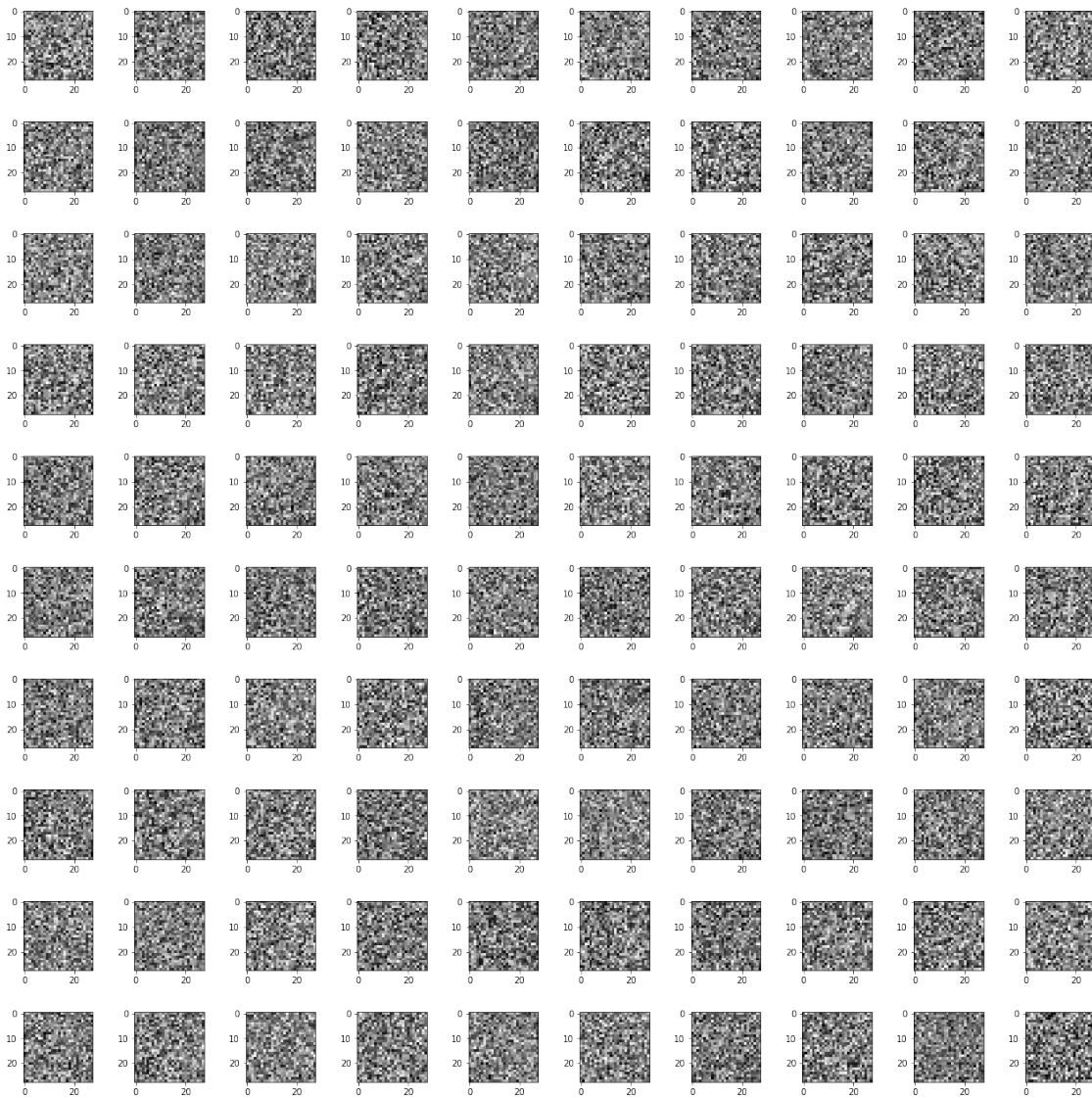
```

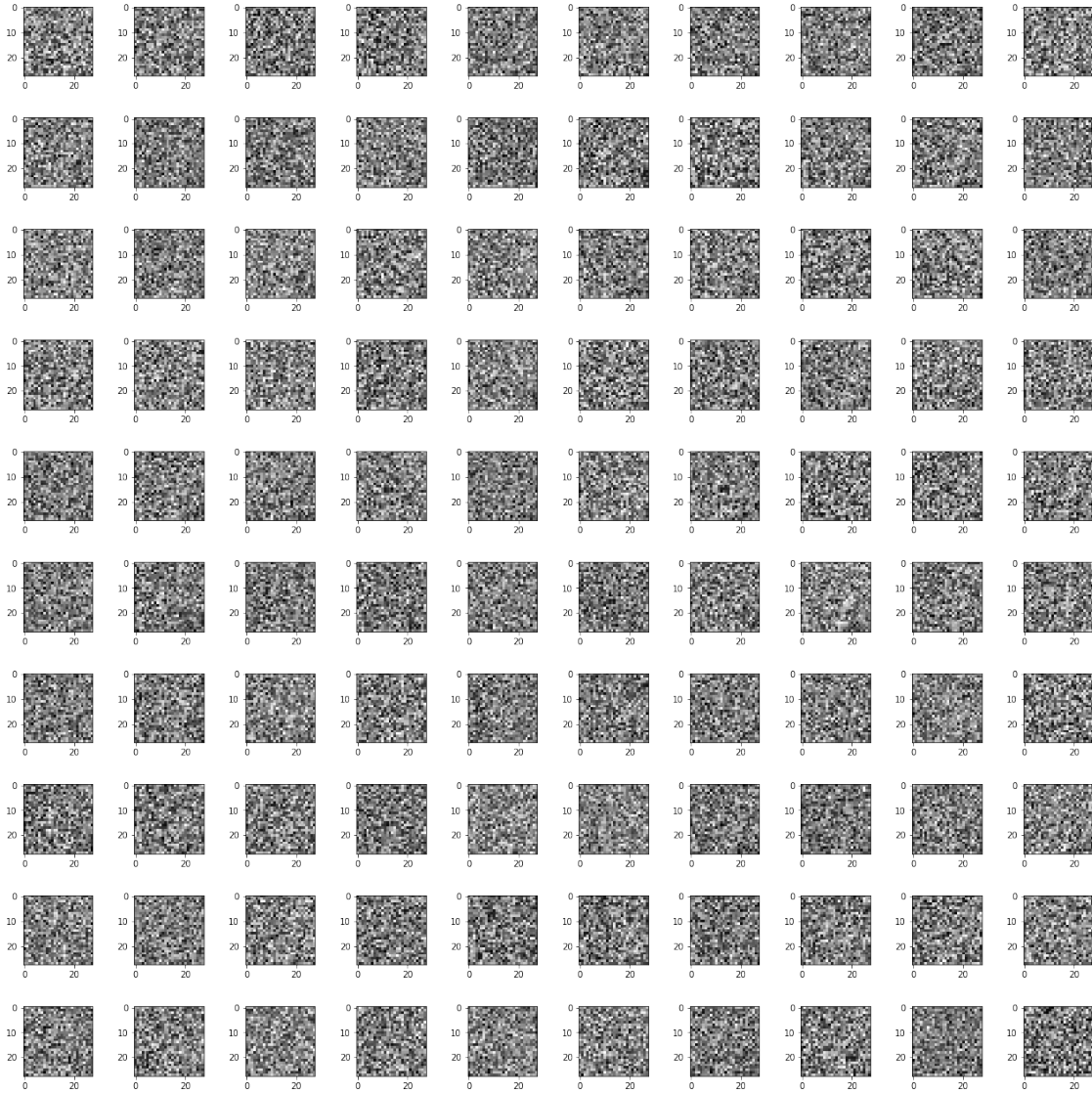
fig.set_figheight(18)
fig.tight_layout()
ax = np.asarray(axes).ravel()
for i in range(len(gen_imgs)):
    ax[i].imshow(gen_imgs_np[i][:][:], cmap='gray', interpolation='none')
fig

(100, 28, 28)
<class 'numpy.ndarray'>

```

Out[13]:





```
In [0]: train()
```

```
In [15]: z_after_train = torch.randn(BATCH_SIZE, GENERATOR_INPUT_SIZE).to(device)
gen_imgs = G(z_after_train).detach().cpu()
gen_imgs_np = gen_imgs.numpy().reshape(100,28,28)
```

```
print(gen_imgs_np.shape)
print(type(gen_imgs_np))
```

```
fig, axes = plt.subplots(nrows=10, ncols=10)
fig.set_figwidth(18)
fig.set_figheight(18)
```



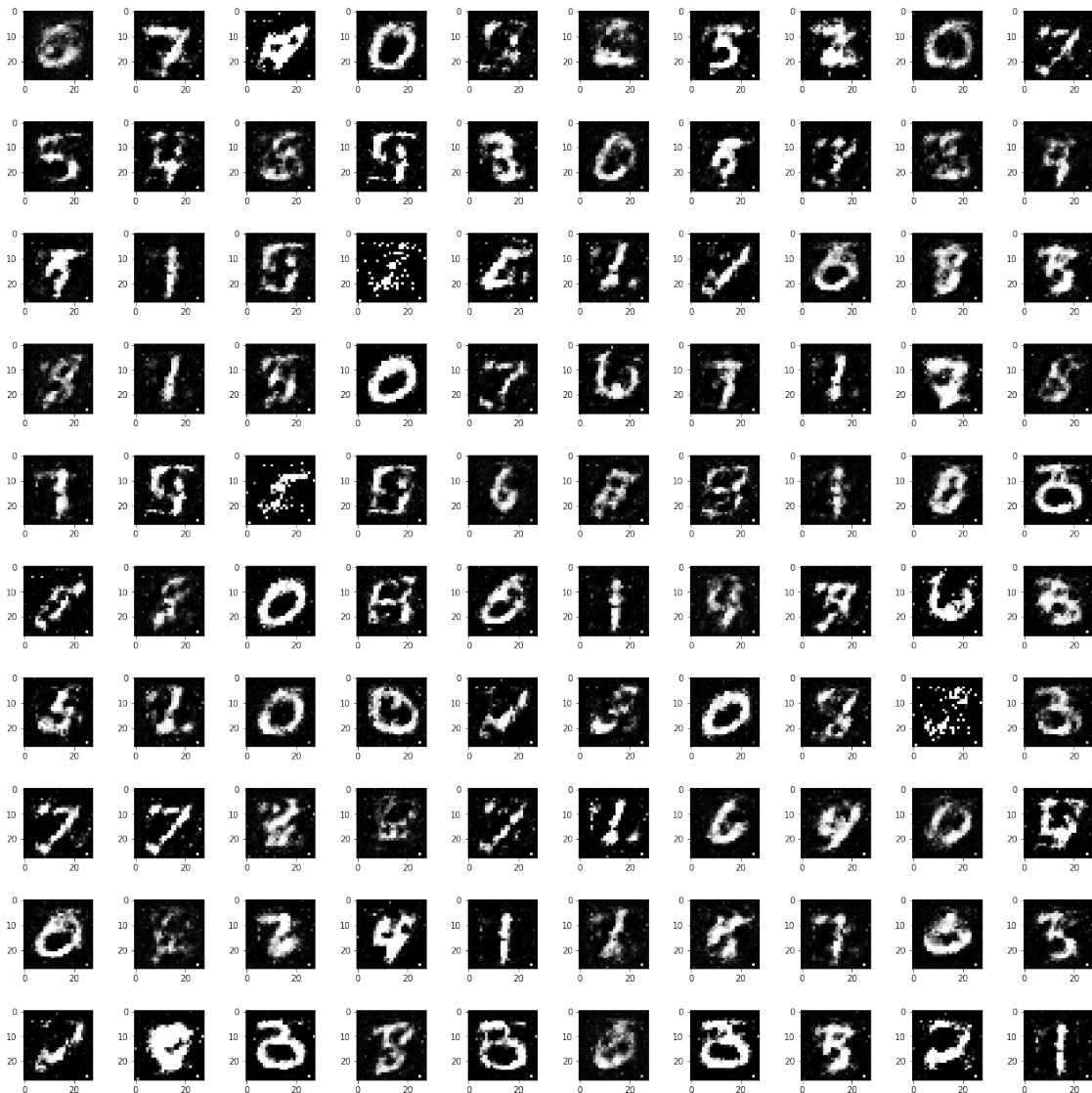
```

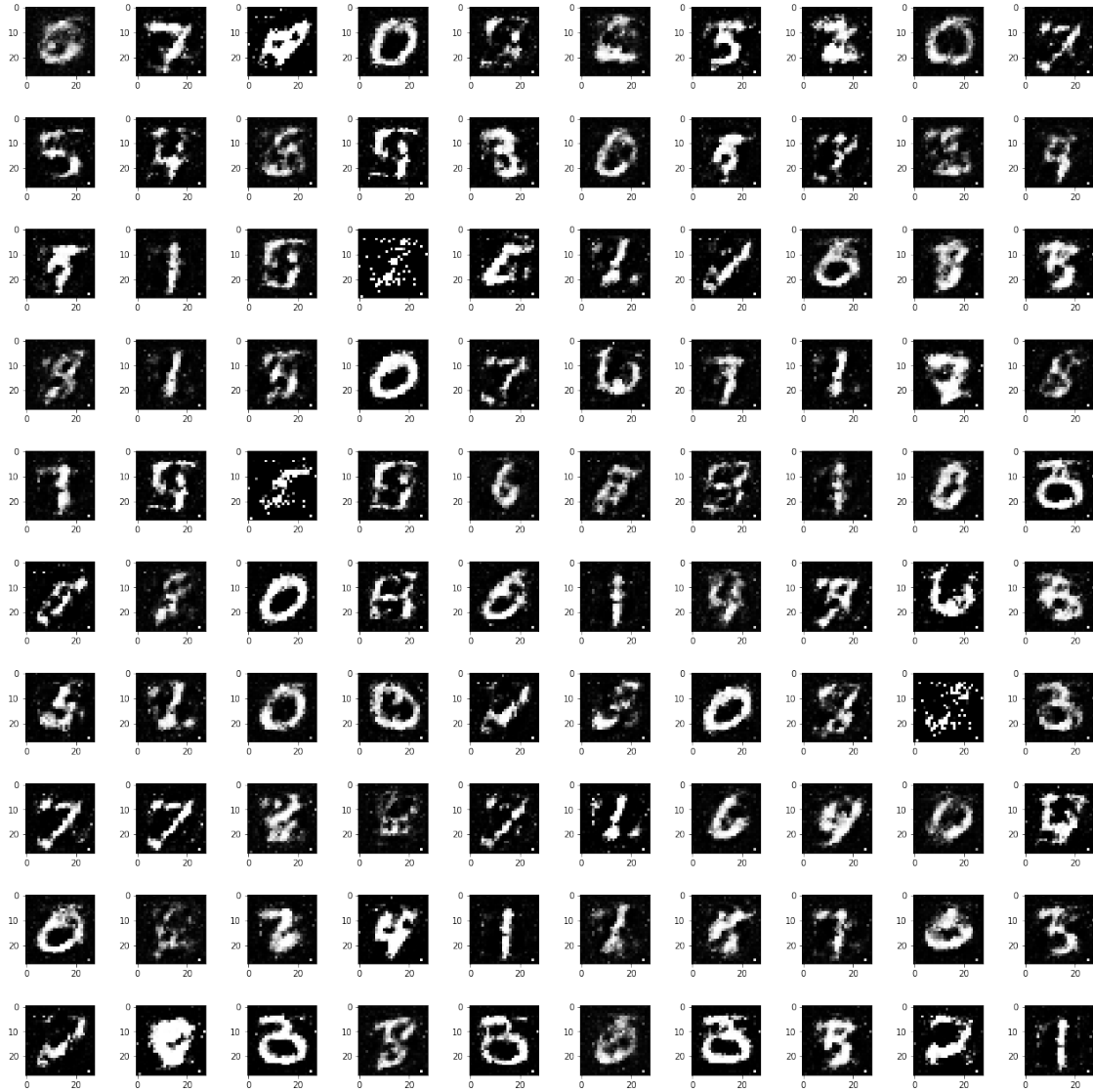
fig.tight_layout()
ax = np.asarray(axes).ravel()
for i in range(len(gen_imgs)):
    ax[i].imshow(gen_imgs_np[i][:][:], cmap='gray', interpolation='none')
fig

(100, 28, 28)
<class 'numpy.ndarray'>

```

Out[15]:





In [0]: # label 0 appears slightly more often. Not sure whether 25 epochs are enough. But there