Building a GAN in PyTorch

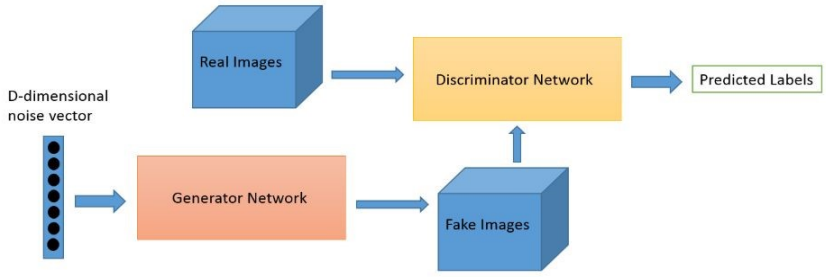
StartOnAI Deep Learning Tutorial 5

This Tutorial covers Generative Adversarial Networks and their PyTorch Implementation

What is a General Adversarial Network?

The **General Adversarial Network** was invented in 2014 by researcher **Ian Godfellow** and his colleagues as a method of employing deep learning in unsupervised learning. Other examples of unsupervised learning algorithms, or clustering techniques include hierarchical clustering, K-means clustering, Principal component analysis, and singular value decomposition. As the name suggests General Adversarial Network or **GAN** uses a "generative" model which has the capability to generate data. **Generative modeling** is a process in unsupervised learning in which a model is trained to generate data by observing features and trends in a natural and realistic dataset. Through excessive training a GAN can generate data that closely resembles data from the supplied data-set. For example, given a large dataset of an individual's voice, it is even possible for the GAN to generate very close replications to that individual's voice. However, the GAN model is not limited to a generative model and also uses a discriminative model. **Discriminative models** are used for classification and function by optimizing the parameters of the model to better fit a dataset with labels. Examples of discriminative modeling include logistic regression, deep neural networks, support vector machines, etc. So how do two completely different types of models work together to generate data? Let's explore the process in the next section.

How does a GAN actually work?



A GAN consists of two structures:

1. **Generator neural network** which intakes random numbers
2. **Discriminator neural network** which intakes a training set and generated data(usually images but can use other types of training data)

In this tutorial we will be discussing the use of a GAN for images.The discriminator neural network(typically a convolutional neural network) intakes images and uses binary classification model for its output. Specifically, it determines if the image was taken from the same distribution as the training data or if it is an imposter image that was generated by the generator network. Mathematically it calculates the probability that the input image was not generated by the generator neural network. The back-propogation algorithm is then used by the discriminator to minimize the error of the cost between the predicted probability and binary classification(0 or 1) for each image in the training set.

On the other hand the generator neural network can essentially be thought of as a complex probability distribution which intakes random numbers and outputs a final image after a forward pass through the network. Each output neuron represents a pixel in the image. The goal of the generator network is to maximize the error of the cost function in the discriminator neural network. Thus both networks are essentially going up against one another and attempting to exploit the others weakness.

If the relationship between the networks is confusing let me bring you an analogy. Imagine you are playing a game of chess against another player. In order to maximize the rate of victory you want to observe other players and improve by exploiting the opponent's weakness. The opponent will also try to do the same. He will observe you and try to formulate a strategy to exploit your weakness. After every game both of you

are successively getting better at playing the game against each other and as a result you will be forced to improve. You and the opponent will be forced to get rid of bad habits while being encouraged to capitalize on one anothers bad habits. In a similar way the generator is trying to produce images that trick the discriminator and the discriminator is trying to improve its authenticity detection.

Typically when training one network, the other is kept static. Another way to think of that is that backpropagation only runs on one network at a time and not simultaneously. While the generator is modifying its parameters the discriminator is only being used for forward propagation and while the discriminator is modifying its parameters the generator is being used to feed images to the discriminator. This allows for a more clear and stable vision of the cost-function when employing back-propagation and allows for a successful convergence of gradient descent. Let's look at an example of applying a GAN in the next section.

What is the MNIST data-set and how will we apply the GAN model?

MNIST stands for "Modified National Institutes of Standards and Technology". Really it's just a very large database of handwritten digits(0-9). It has around 60,000 images and each image is 28x28 pixels. We will be writing some code in pytorch to train a GAN to generate images resembling those of MNIST.

Code walkthrough(code from Tanay Agarwal https://github.com/tanayag/gans):

```
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
from torch.autograd import Variable
import pickle
from torch.utils import data as t_data
import torchvision.datasets as datasets
from torchvision import transforms
%matplotlib inline
```

Firstly we need to take care of all the necessary imports. The main ones are numpy, torch and matplotlib(for visualization). We also should import specific models from py torch to access our dataset and also to make coding a little easier.

```
data_transforms = transforms.Compose([transforms.ToTensor()])
```

```
mnist_trainset = datasets.MNIST(root='./data', train=True, download=True, transform=data_transforms)
```

```
mnist_trainset
```

The first step to any machine learning problem is retrieving the data so lets do that.

```
# defining generator class

class generator(nn.Module):

    def __init__(self, inp, out):

        super(generator, self).__init__()

        self.net = nn.Sequential(
                                nn.Linear(inp,300),
                                nn.ReLU(inplace=True),
                                nn.Linear(300,1000),
                                nn.ReLU(inplace=True),
                                nn.Linear(1000,800),
                                nn.ReLU(inplace=True),
                                nn.Linear(800,out)
                                   )

    def forward(self, x):
        x = self.net(x)
        return x
```

Fortunately for us the MNIST is a commonly used dataset so we don't need to write more than a few lines of code to import it. While loading we will also normalize the data. Now we have an object in pytorch which will store our training data in the dataloader_mnist_train variable.

```
def make_some_noise():
    return torch.rand(batch_size,100)
```

This is a simple helper function that will allow us to provide our generator network with a vector of "noise" or random numbers.

```
# defining discriminator class

class discriminator(nn.Module):

    def __init__(self, inp, out):

        super(discriminator, self).__init__()

        self.net = nn.Sequential(
                                nn.Linear(inp,300),
                                nn.ReLU(inplace=True),
                                nn.Linear(300,300),
                                nn.ReLU(inplace=True),
                                nn.Linear(300,200),
                                nn.ReLU(inplace=True),
                                nn.Linear(200,out),
                                nn.Sigmoid()
                                   )

    def forward(self, x):
        x = self.net(x)
        return x
```

Fortunately when defining our generator class we can inherit from nn.Module. We only need to do two things. Firstly we should build our Sequential model for the neural network. In this case we are using 3 hidden layers(excluding output layer) with 300, 1000 and 800 hidden neurons respectively. Our activation function is RELU. We should also make a simple function **forward()** that returns an output given an input noise vector. We do the same thing for the discriminator except we use a slightly different neural network architecture. We now have 3 hidden layers(excluding the output) with 300,300,200 hidden units respectively. This time we use the sigmoid activation function for the output layer since this network is outputting a binary classification.

```
d_steps = 1
g_steps = 1

criteriond1 = nn.BCELoss()
optimizerd1 = optim.SGD(dis.parameters(), lr=0.001, momentum=0.9)

criteriond2 = nn.BCELoss()
optimizerd2 = optim.SGD(gen.parameters(), lr=0.001, momentum=0.9)

printing_steps = 200

epochs = 4000
```

Before we go into training lets define some variables. First of all we want to use binary cross entropy loss for both of our networks as we are using binary classification. Our

optimization algorithm will be stochastic gradient descent so let's set up optimizers for both of our networks with the appropriate learning rate, parameters and momentum.

```
for epoch in range(epochs):

    if epoch%printing_steps==0:
        print epoch

    # training discriminator
    for d_step in range(d_steps):
        dis.zero_grad()

        # training discriminator on real data
        for inp_real,_ in dataloader_mnist_train:
            inp_real_x = inp_real
            break

        inp_real_x = inp_real_x.reshape(4,784)
        dis_real_out = dis(inp_real_x)
        dis_real_loss = criteriond1(dis_real_out,Variable(torch.ones(batch_size,1)))
        dis_real_loss.backward()

        # training discriminator on data produced by generator
        inp_fake_x_gen = make_some_noise()
        dis_inp_fake_x = gen(inp_fake_x_gen).detach() #output from generator is generated
        dis_fake_out = dis(dis_inp_fake_x)
        dis_fake_loss = criteriond1(dis_fake_out,Variable(torch.zeros(batch_size,1)))
        dis_fake_loss.backward()

        optimizerd1.step()



    # training generator
    for g_step in range(g_steps):
        gen.zero_grad()

        #generating data for input for generator
        gen_inp = make_some_noise()

        gen_out = gen(gen_inp)
        dis_out_gen_training = dis(gen_out)
        gen_loss = criteriond2(dis_out_gen_training,Variable(torch.ones(batch_size,1)))
        gen_loss.backward()
```

Let us go through this training process step by step:

1. Our outer loop represents iterating over all epochs
2. Train the discriminator on the data from the training set
3. Calculate loss and call back propagation on our initialized optimizer
4. Train the discriminator on data generated by the generator.
5. Start off by providing noise to the generator. Then train the generator by producing multiple images, feeding it to the discriminator, calculating loss, and then back propagating.

More Resources: https://developers.google.com/machine-learning/gan/gan_structure
https://www.youtube.com/watch?v=5g1eXmQtl0E&t=362s
https://www.youtube.com/watch?v=9JpdAg6uMXs&t=555s
https://venturebeat.com/2019/12/26/gan-generative-adversarial-network-explainer-ai-ma chine-learning/ https://machinelearningmastery.com/what-are-generative-adversarial-networks-gans/

Citations

Agrawal, Tanay. "Train Your First GAN Model from Scratch Using PyTorch." *Medium*, Noteworthy - The Journal Blog, 7 July 2019, blog.usejournal.com/train-your-first-gan-model-from-scratch-using-pytorch-9b7298 7fd2c0.

"A Beginner's Guide to Generative Adversarial Networks (GANs)." *Pathmind*, pathmind.com/wiki/generative-adversarial-network-gan.

Brownlee, Jason. "A Gentle Introduction to Generative Adversarial Networks (GANs)." *Machine Learning Mastery*, 19 July 2019, machinelearningmastery.com/what-are-generative-adversarial-networks-gans/.

"GAN Model." *A Beginner's Guide to Generative Adversarial Networks (GANs)*, pathmind.com/wiki/generative-adversarial-network-gan.

"The MNIST Database." *Pathmind*, pathmind.com/wiki/mnist.