Simple-GAN-v1.0

November 7, 2021

1 Simplest GAN with PyTorch!

In this notebook we are going to implement simplest GAN possible! The chosen library for this task is PyTorch. A misunderstanding regarding the GANs are a false belief that they can only produce pictures. this is completely wrong, they can produce anything! in this notebook we are going to train a GAN which will produce odd numbers!

[ ]:
```
import math
import numpy as np
import torch
import torch.nn as nn
```

1.1 Useful functions and creating the data

our first task is to make number binary and save each digit as list. with this approach we can check whether if a number is even or odd by looking at its right digit. also we can pass binary vectors to neural nets without difficulties.

[ ]:
```
def convert_binary_list(number):
    if type(number) is not int or number < 0:
        raise ValueError('Enter Positive Integer!')

    return [int(x) for x in list(bin(number))[2:]]
```

now lets implement another function to generate odd number for training!

[ ]:
```
def generate_odd_number(max_int, batch_size = 16):
    max_length = math.ceil(math.log(max_int, 2))                # calculating␣
↪the maximum length of possible numbers in binary
    samples = []
    while len(samples) != batch_size:
        num = np.random.randint(0, max_int)
        samples.append(num) if num % 2 != 0 else next           # adding number␣
↪to list if that number is odd
    labels = [1] * batch_size                                   # generating 1␣
↪as label for every number, 1 since all of them are odd already.
    data = [convert_binary_list(x) for x in samples]            # converting␣
↪generated odd numbers to binary
```

```
    data = [([0] * (max_length - len(x))) + x for x in data]    # padding all␣
↪of the number to same length by adding 0 to right digits

    return labels, data
```

[ ]:
```
def convert_float_matrix_to_int_list(float_matrix, threshold = .5):
    return [int("".join([str(int(y)) for y in x]), 2) for x in float_matrix >=␣
↪threshold]
```

1.2 Generator

now lets make the generator! Since our task is fairly easy, a simple linear layer with sigmoid activation can do the job and there are no need to make this neural net sophisticated. note that for more complicated tasks, like generating sceneries, we might have to use more complicated neural nets.

[ ]:
```
class Generator(nn.Module):
    def __init__(self, input_len):
        super(Generator, self).__init__()
        self.linear1 = nn.Linear(int(input_len), int(input_len))
        self.activation1 = nn.Sigmoid()

    def forward(self, x):
        out = self.linear1(x)
        out = self.activation1(out)
        return out
```

1.3 Discriminator

the same thing goes for the discriminator too, it's not more complicate than generator. this part takes a binary number as input and checks whether that number is "original" (odd) or "fake" (even). since this task is fairly simple and easy, a linear layer with sigmoid activation can do this operation so there is no obligation to develope more complicated model.

[ ]:
```
class Discriminator(nn.Module):
    def __init__(self, input_len):
        super(Discriminator, self).__init__()
        self.linear1 = nn.Linear(int(input_len), 1)
        self.activation1 = nn.Sigmoid()

    def forward(self, x):
        out = self.linear1(x)
        out = self.activation1(out)
        return out
```

1.4 Training the GAN

i believe training is the trickest part of GAN. in this section we have too link the generator and discrimator and train them in unison. the reason behind this linkeage is correct propagating the gradients so the generator can "learn".

at every training step in GAN we need 2 batches of data, one is random noise for generator to create a new number and second batch is our "original" data.

[ ]:
```
def train(max_int = 128, batch_size = 16, epoch = 1000, print_output_n_steps =␣
↪10):
    input_length = math.ceil(math.log(max_int, 2))

    generator = Generator(input_length)
    discriminator = Discriminator(input_length)

    generator_optimizer = torch.optim.Adam(generator.parameters(), lr = .001)
    discriminator_optimizer = torch.optim.Adam(discriminator.parameters(), lr =␣
↪.001)

    loss = nn.BCELoss()

    for i in range(epoch):
        generator_optimizer.zero_grad()                                    ␣
↪                                       # we have to zero gradients for each␣
↪iteration.

        noise = torch.randint(0, 2, size=(batch_size, input_length)).float()  ␣
↪                                       # creating noise for generator, Have␣
↪to be Float, not int.
        generated_data = generator(noise)

        org_label, org_data = generate_odd_number(max_int, batch_size)     ␣
↪                                       # creating original data
        org_label = torch.tensor(org_label).float()
        org_label = org_label.unsqueeze(1)                                 ␣
↪                                       # torch.size(16) and torch.size([16,␣
↪1]) are no longer equal
        org_data = torch.tensor(org_data).float()

        generator_discriminator_out = discriminator(generated_data)        ␣
↪                                       # training generator
        generator_loss = loss(generator_discriminator_out, org_label)
        generator_loss.backward()
        generator_optimizer.step()

        discriminator_optimizer.zero_grad()                                ␣
↪                                       # training discriminator
```

```
        org_discriminator_out = discriminator(org_data)
        org_discriminator_loss = loss(org_discriminator_out, org_label)

        generator_discriminator_out = discriminator(generated_data.detach())
↪                                       # dont forget to detach
        generator_discriminator_loss = loss(generator_discriminator_out, torch.
↪zeros(batch_size).unsqueeze(1))
        discriminator_loss = (org_discriminator_loss +
↪generator_discriminator_loss) / 2
        discriminator_loss.backward()
        discriminator_optimizer.step()

        int_generated_data = convert_float_matrix_to_int_list(generated_data)
↪                                       # converting generated data to int
↪and calculating accuracy
        generated_data_even_count = len([num for num in int_generated_data if
↪num % 2 == 0])
        error = (generated_data_even_count / batch_size) * 100

        if i % print_output_n_steps == 0:
            print(f'Step: {i}/{epoch}, Error: {error:.1f}%, Generator Loss:
↪{generator_loss.item():.4f}, Discriminator Loss: {discriminator_loss.item():.
↪4f}, Overall loss: {generator_discriminator_loss.item():.4f}')
            print(f'Sample: \t{int_generated_data} \n')

    return generator, discriminator
```

[ ]:
```
train(epoch=1001, print_output_n_steps=100, batch_size=16)
```

```
Step: 0/1001, Error: 50.0%, Generator Loss: 0.8213, Discriminator Loss: 0.7348,
Overall loss: 0.5798
Sample:         [6, 85, 31, 94, 84, 95, 91, 95, 84, 31, 86, 15, 15, 92, 90, 92]

Step: 100/1001, Error: 93.8%, Generator Loss: 0.6721, Discriminator Loss:
0.7372, Overall loss: 0.7150
Sample:         [68, 116, 124, 70, 76, 76, 68, 110, 85, 116, 68, 68, 68, 100,
108, 108]

Step: 200/1001, Error: 100.0%, Generator Loss: 0.6647, Discriminator Loss:
0.7052, Overall loss: 0.7225
Sample:         [100, 100, 102, 102, 100, 110, 102, 100, 100, 100, 100, 100,
102, 102, 100, 116]

Step: 300/1001, Error: 93.8%, Generator Loss: 0.7031, Discriminator Loss:
0.6849, Overall loss: 0.6834
Sample:         [100, 102, 102, 116, 100, 102, 102, 102, 102, 101, 102, 102,
102, 102, 114, 102]
```

```
Step: 400/1001, Error: 93.8%, Generator Loss: 0.7133, Discriminator Loss:
0.6585, Overall loss: 0.6735
Sample:         [68, 66, 82, 70, 70, 80, 70, 70, 70, 7, 70, 68, 98, 70, 82, 70]

Step: 500/1001, Error: 0.0%, Generator Loss: 0.6859, Discriminator Loss: 0.6687,
Overall loss: 0.7005
Sample:         [81, 81, 81, 81, 65, 89, 65, 81, 65, 83, 81, 83, 83, 81, 81, 81]

Step: 600/1001, Error: 0.0%, Generator Loss: 0.6688, Discriminator Loss: 0.6829,
Overall loss: 0.7182
Sample:         [81, 9, 81, 121, 121, 57, 89, 17, 25, 25, 33, 33, 25, 17, 25,
113]

Step: 700/1001, Error: 0.0%, Generator Loss: 0.6800, Discriminator Loss: 0.6773,
Overall loss: 0.7066
Sample:         [57, 57, 57, 57, 57, 57, 49, 57, 57, 49, 57, 57, 57, 49, 49, 49]

Step: 800/1001, Error: 0.0%, Generator Loss: 0.6998, Discriminator Loss: 0.6686,
Overall loss: 0.6871
Sample:         [43, 41, 59, 59, 59, 11, 27, 43, 43, 59, 59, 43, 59, 49, 49, 57]

Step: 900/1001, Error: 0.0%, Generator Loss: 0.6865, Discriminator Loss: 0.6834,
Overall loss: 0.7002
Sample:         [15, 15, 15, 27, 15, 15, 15, 15, 15, 15, 11, 43, 15, 11, 11, 11]

Step: 1000/1001, Error: 0.0%, Generator Loss: 0.6964, Discriminator Loss:
0.6770, Overall loss: 0.6900
Sample:         [7, 15, 31, 15, 7, 7, 15, 7, 7, 7, 7, 7, 15, 15, 15, 7]
```

[ ]:
```
(Generator(
    (linear1): Linear(in_features=7, out_features=7, bias=True)
    (activation1): Sigmoid()
  ),
  Discriminator(
    (linear1): Linear(in_features=7, out_features=1, bias=True)
    (activation1): Sigmoid()
  ))
```

---

2 By Ramin F. | @SimplyRamin