Министерство образования и науки РФ
Санкт-Петербургский Политехнический университет Петра Великого
Институт компьютерных наук и технологий
Высшая школа искусственного интеллекта

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ

«GAN»

по дисциплине «Машинное обучение»

Выполнили:

студентки гр. 3540201/20301 _____ Климова О. А.

_____ Сысоева П. А.

подпись, дата

Проверил:

д.т.н., проф. _____ Уткин Л. В.

подпись, дата

Санкт-Петербург

2023

Содержание

1. Теория

Генеративно-состязательные сети (Generative Adversarial Networks) - алгоритм машинного обучения без учителя, построенный на комбинации из двух нейронных сетей, одна из которых генерирует образцы (генератор), а другая (дискриминатор) старается отличить правильные («подлинные») образцы от неправильных (рис. 1).

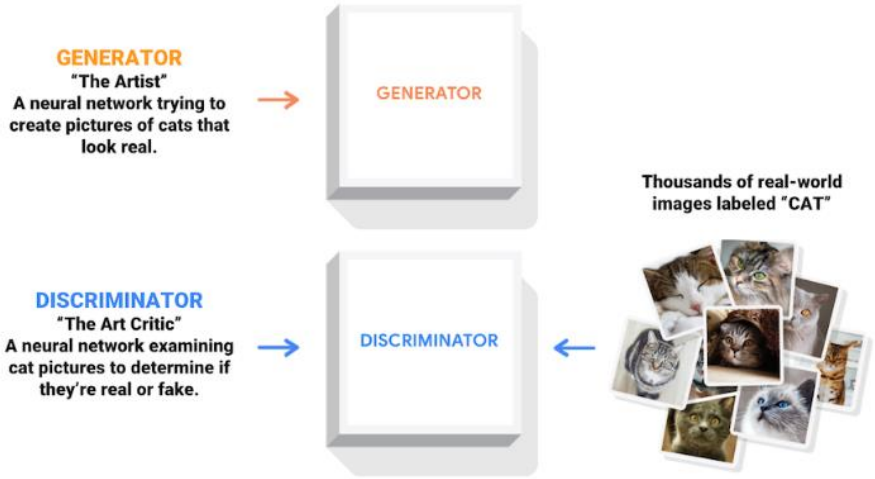


Рисунок 1 – Generator and Discriminator

Во время обучения генератор постепенно становится лучше в создании изображений, которые выглядят реальными, в то время как дискриминатор лучше различает их. Процесс достигает равновесия, когда дискриминатор уже не может отличить настоящие изображения от подделок (рис. 2).

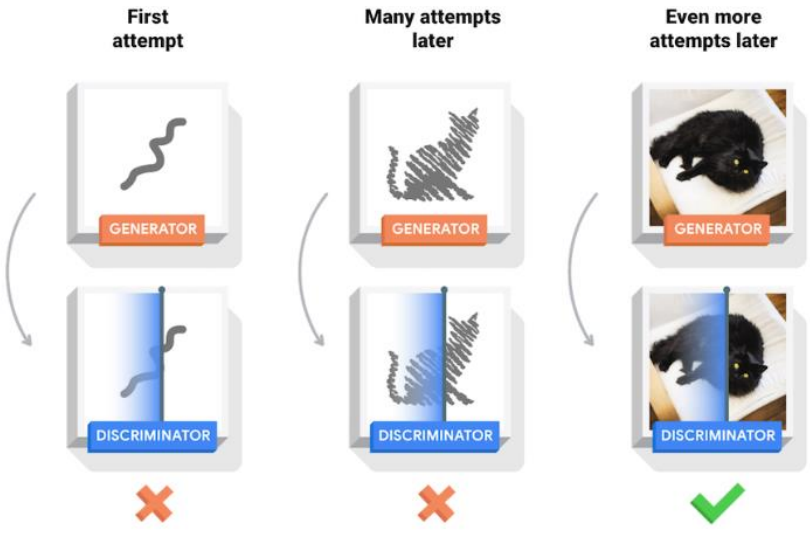


Рисунок 2 – Достижение равновесия

2. Пример работы с GAN

Демонстрация работы с генеративно-состязательными сетями производится с использованием набора данных MNIST (рис. 3).

Рисунок 3 – Набор данных MNIST

Изображения начинаются как случайный шум и со временем все больше напоминают рукописные цифры. Обучение генератора производится в течение 100 эпох (рис. 4).

Рисунок 3 – Результат работы GAN

Во время обучения мы стремимся максимизировать вероятность правильной идентификации объектов log(D(x|y)) и минимизировать вероятность того, что дискриминатор примет нереальную картинку за нереальную log(1−D(G(z))). Таким образом, генератор и дискриминатор играют в «минимакс игру»:

$$\min_G \max_D \underset{x \sim p_{data}}{E} [logD(x|y)] + \underset{z \sim p_z}{E} [log(1 - D(G(z|y))]$$

Функция потерь для GAN представляет из себя совокупность функций потерь генератора и дискриминатора (рис. 4).

Рисунок 4 – Функции потерь генератора и дискриминатора

Код, демонстрирующий работу с GAN представлен далее:

```
%matplotlib inline

import numpy as np
import torch
import matplotlib.pyplot as plt
from torchvision import datasets
import torchvision.transforms as transforms
```

Определяем параметры для обучения модели:

```
num_workers = 0
batch_size = 64
```

```
transform = transforms.ToTensor()

# импортируем набор данных для обучения
train_data = datasets.MNIST(root='data', train=True,
                                    download=True, transform=transform)
# определяем загрузчик данных
train_loader = torch.utils.data.DataLoader(train_data,
batch_size=batch_size, num_workers=num_workers)
```

Дискриминатор:

```
import torch.nn as nn
import torch.nn.functional as F

class Discriminator(nn.Module):

    def __init__(self, input_size, hidden_dim, output_size):
        super(Discriminator, self).__init__()

        # определяем скрытые слои
        self.fc1 = nn.Linear(input_size, hidden_dim*4)
        self.fc2 = nn.Linear(hidden_dim*4, hidden_dim*2)
        self.fc3 = nn.Linear(hidden_dim*2, hidden_dim)
        # последний полносвязный слой
        self.fc4 = nn.Linear(hidden_dim, output_size)

        self.dropout = nn.Dropout(0.3)


    def forward(self, x):
        # переводим в вектор
        x = x.view(-1, 28*28)

        # скрытые слои
        x = F.leaky_relu(self.fc1(x), 0.2) # (input, negative_slope=0.2)
        x = self.dropout(x)
        x = F.leaky_relu(self.fc2(x), 0.2)
        x = self.dropout(x)
        x = F.leaky_relu(self.fc3(x), 0.2)
        x = self.dropout(x)
        # финальный слой
        out = self.fc4(x)

        return out
```

Генератор:

```
class Generator(nn.Module):

    def __init__(self, input_size, hidden_dim, output_size):
        super(Generator, self).__init__()

        # определяем скрытые слои
        self.fc1 = nn.Linear(input_size, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim*2)
        self.fc3 = nn.Linear(hidden_dim*2, hidden_dim*4)
        # финальный полносвязный слон
        self.fc4 = nn.Linear(hidden_dim*4, output_size)

        self.dropout = nn.Dropout(0.3)

    def forward(self, x):
        # скрытые слои
        x = F.leaky_relu(self.fc1(x), 0.2) # (input, negative_slope=0.2)
        x = self.dropout(x)
        x = F.leaky_relu(self.fc2(x), 0.2)
        x = self.dropout(x)
        x = F.leaky_relu(self.fc3(x), 0.2)
        x = self.dropout(x)
        # финальный слой
        out = F.tanh(self.fc4(x))

        return out
```

Гиперпараметры дискриминатора и генератора:

```
# Дискриминатор
input_size = 784
d_output_size = 1
d_hidden_size = 32

# Генератор
z_size = 100
g_output_size = 784
g_hidden_size = 32
```

Определяем модели:

```
D = Discriminator(input_size, d_hidden_size, d_output_size)
G = Generator(z_size, g_hidden_size, g_output_size)
```

Функции потерь для генератора и дискриминатора:

```
def real_loss(D_out, smooth=False):
```

```
    batch_size = D_out.size(0)
    if smooth:
        labels = torch.ones(batch_size)*0.9
    else:
        labels = torch.ones(batch_size)

    criterion = nn.BCEWithLogitsLoss()

    loss = criterion(D_out.squeeze(), labels)
    return loss

def fake_loss(D_out):
    batch_size = D_out.size(0)
    labels = torch.zeros(batch_size)
    criterion = nn.BCEWithLogitsLoss()

    loss = criterion(D_out.squeeze(), labels)
    return loss
```

Создание оптимизаторов для дискриминатора и генератора:

```
import torch.optim as optim

# Коэффициент скорости обучения
lr = 0.002
d_optimizer = optim.Adam(D.parameters(), lr)
g_optimizer = optim.Adam(G.parameters(), lr)
```

Обучение генератора и дискриминатора:

```
import pickle as pkl

# обучающий гиперпараметр
num_epochs = 100
samples = []
losses = []
print_every = 400

# Получение фиксированных данных для выборки
# они постоянно хранятся и позволяют отслеживать производительность модели
sample_size=16
fixed_z = np.random.uniform(-1, 1, size=(sample_size, z_size))
fixed_z = torch.from_numpy(fixed_z).float()

# обучение
D.train()
G.train()
```

```
for epoch in range(num_epochs):

    for batch_i, (real_images, _) in enumerate(train_loader):
        batch_size = real_images.size(0)

        #масштабирование входных изображений из [0,1) в [-1, 1)
        real_images = real_images*2 - 1

        #ОБУЧЕНИЕ ДИСКРИМИНАТОРА
        d_optimizer.zero_grad()

        # 1. Обучение с реальными картинками
        # Подсчет потерь с реальными картинками
        D_real = D(real_images)
        d_real_loss = real_loss(D_real, smooth=True)

        # 2. Обучение с фэйковыми картинками
        # Генерация фэйковых картинок
        z = np.random.uniform(-1, 1, size=(batch_size, z_size))
        z = torch.from_numpy(z).float()
        fake_images = G(z)
        # Подсчет потерь с фэйковыми картинками
        D_fake = D(fake_images)
        d_fake_loss = fake_loss(D_fake)

        # суммирование функций потерь
        d_loss = d_real_loss + d_fake_loss
        d_loss.backward()
        d_optimizer.step()


        #ОБУЧЕНИЕ ГЕНЕРАТОРА
        g_optimizer.zero_grad()

        # 1. Обучение с фэйковыми картинками
        # генерация фэйковых картинок
        z = np.random.uniform(-1, 1, size=(batch_size, z_size))
        z = torch.from_numpy(z).float()
        fake_images = G(z)

        # Подсчет потерь с фэйковыми картинками
        D_fake = D(fake_images)
        g_loss = real_loss(D_fake)

        g_loss.backward()
        g_optimizer.step()

        # Выводим потери для дискриминатора и генератора
        if batch_i % print_every == 0:
```

```
            print('Epoch [{:5d}/{:5d}] | d_loss: {:6.4f} | g_loss:
{:6.4f}'.format(epoch+1, num_epochs, d_loss.item(), g_loss.item()))

    # объединение потерь дискриминатора и генератора
    losses.append((d_loss.item(), g_loss.item()))

    # сохраняем образцы фэйковых изображений
    G.eval()
    samples_z = G(fixed_z)
    samples.append(samples_z)
    G.train()

# сохраняем образцы обучения генератора
with open('train_samples.pkl', 'wb') as f:
    pkl.dump(samples, f)
```

Построение функций потерь:

```
fig, ax = plt.subplots()
losses = np.array(losses)
plt.plot(losses.T[0], label='Discriminator')
plt.plot(losses.T[1], label='Generator')
plt.title("Training Losses")
plt.legend()
```

Вспомогательная функция для вывода изображений:

```
def view_samples(epoch, samples):
    fig, axes = plt.subplots(figsize=(7,7), nrows=4, ncols=4, sharey=True,
sharex=True)
    for ax, img in zip(axes.flatten(), samples[epoch]):
        img = img.detach()
        ax.xaxis.set_visible(False)
        ax.yaxis.set_visible(False)
        im = ax.imshow(img.reshape((28,28)), cmap='Greys_r')
```

Вывод полученных результатов:

```
# Скачивание образцов из генератора
with open('train_samples.pkl', 'rb') as f:
    samples = pkl.load(f)

view_samples(-1, samples)
rows = 10
cols = 6
fig, axes = plt.subplots(figsize=(7,12), nrows=rows, ncols=cols,
sharex=True, sharey=True)

for sample, ax_row in zip(samples[::int(len(samples)/rows)], axes):
```

```
    for img, ax in zip(sample[::int(len(sample)/cols)], ax_row):
        img = img.detach()
        ax.imshow(img.reshape((28,28)), cmap='Greys_r')
        ax.xaxis.set_visible(False)
        ax.yaxis.set_visible(False)
```

Тестирование (inference):

```
# Случайно генерируем новые изображения
sample_size=16
rand_z = np.random.uniform(-1, 1, size=(sample_size, z_size))
rand_z = torch.from_numpy(rand_z).float()

G.eval()
rand_images = G(rand_z)

# Выводим полученные изображения, которые были сгенерированы обученным
генератором
view_samples(0, [rand_images])
```

3. Задания

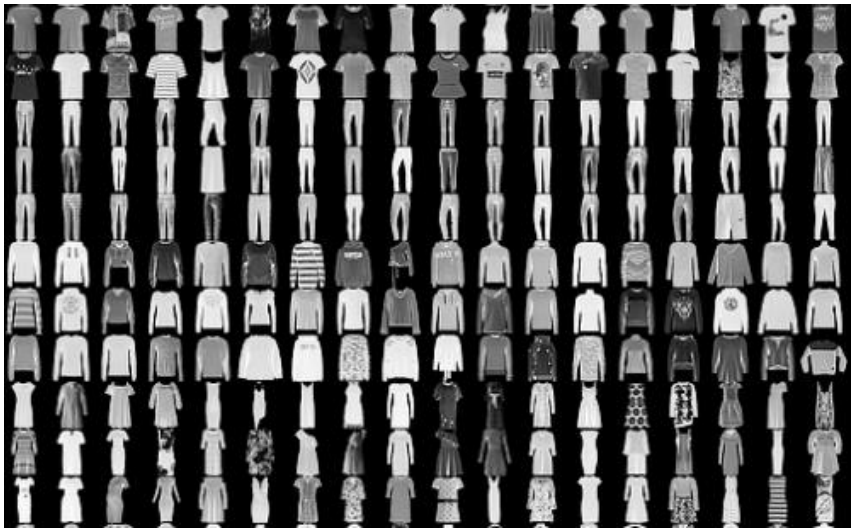
3.1 Задание № 1

Реализовать модель на том же наборе данных MNIST:

Проверить работу модели с различным числом эпох: 50, 150, 200. Как изменяется качество генерации изображений относительно описанного примера? Сколько времени занимает обучение модели?

3.2 Задание № 2

Реализовать модель наборе данных Fashion - MNIST:



Проверить работу модели с различным числом эпох: 50, 100. Как изменяется качество генерации изображений? Сколько времени занимает обучение модели?

Loading Fashion - MNIST data with Tensorflow

Способ № 1

```
from tensorflow.examples.tutorials.mnist import input_data
data = input_data.read_data_sets('data/fashion')

data.train.next_batch(BATCH_SIZE)
```

Способ № 2 (через url)

```
data = input_data.read_data_sets('data/fashion', source_url='http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/')
```

Способ № 3 (скачать из официального руководства Tensorflow)

https://www.tensorflow.org/tutorials/keras/classification?hl=ru

3.3 Задание № 3

Реализовать модель наборе данных KMNIST:

Проверить работу модели с различным числом эпох: 50, 100. Как изменяется качество генерации изображений? Сколько времени занимает обучение модели?

Loading KMNIST

https://github.com/rois-codh/kmnist