Maksym Shevchenko, AI-1

Laboratory work 1

DeepFake Generation using DCGAN

Theory

Deep fake (also spelled deepfake) is a type of artificial intelligence used to create convincing images, audio and video hoaxes. The term, which describes both the technology and the resulting bogus content, is a portmanteau of deep learning and fake.

Deepfake content is created by using two competing AI algorithms -- one is called the generator and the other is called the discriminator. The generator, which creates the phony multimedia content, asks the discriminator to determine whether the content is real or artificial.

Together, the generator and discriminator form something called a generative adversarial network (GAN). Each time the discriminator accurately identifies content as being fabricated, it provides the generator with valuable information about how to improve the next deepfake.

The first step in establishing a GAN is to identify the desired output and create a training dataset for the generator. Once the generator begins creating an acceptable level of output, it can be fed to the discriminator.

As the generator gets better at creating fake images, the discriminator gets better at spotting them. Conversely, as the discriminator gets better at spotting fake image, the generator gets better at creating them.

A Deep Convolutional GAN (DCGAN) is a direct extension of the GAN described above, except that it explicitly uses convolutional and convolutional-transpose layers in the discriminator and generator, respectively. The discriminator is made up of strided convolution layers, batch norm layers, and LeakyReLU activations. The input is a RGB input image and the output is a scalar probability that the input is from the real data distribution. The generator is comprised of convolutional-transpose layers, batch norm layers, and ReLU activations. The input is a latent vector, that is drawn from a standard normal distribution and the output is a RGB image. The strided conv-transpose layers allow the latent vector to be transformed into a volume with the same shape as an image.

For more detailed information about all layers I used in this work it's better to read official TensorFlow documentation: https://www.tensorflow.org/api_docs/python/tf/keras/layers.

Dataset

The first and main dataset for my work is MNIST database of handwritten digits, available from this page, has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image. It is a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing and formatting.

Second dataset I used, but without good results, is Street View House Number (SVHN) is a digit classification benchmark dataset that contains 600000 32×32 RGB images of printed digits (from 0 to 9) cropped from pictures of house number plates. The cropped images are centered in the digit of interest, but nearby digits and other distractors are kept in the image. SVHN has three sets: training, testing sets and an extra set with 530000 images that are less difficult and can be used for helping with the training process.

Implementation

All code you can find here: https://github.com/Gurdel/Magistracy/tree/main/Machine%20Learning. Notebooks with last DCGAN models are gan_last.ipynb (MNIST) and gan_color.ipynb (AVHN). Other files named *gan* are previous version with another GAN models.

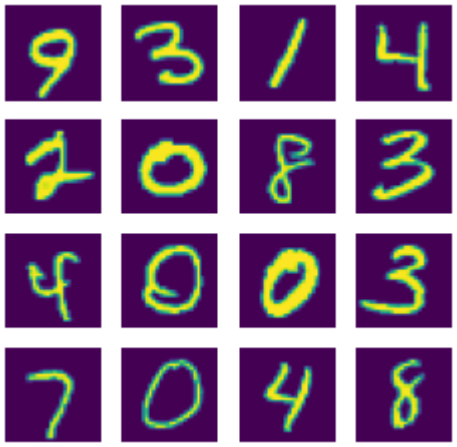
1) Import necessary modules.

```
%matplotlib notebook
%matplotlib inline

import tensorflow as tf
import numpy as np
import random

from matplotlib import pyplot as plt
from tensorflow.keras.layers import Dense, Flatten, Conv2D, BatchNormalization
from tensorflow.keras.layers import Conv2DTranspose, Reshape, LeakyReLU
from tensorflow.keras.models import Model, Sequential
from time import time
```

2) Download dataset using builded-in tf.keras loader and concatenate test and training data for getting more real image examples. Then we need to normalize data to range [0, 1], because generator generates values in this range.

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
x = np.concatenate([x_train, x_test], axis=0)
y = np.concatenate([y_train, y_test], axis=0)

x = x / 255
```

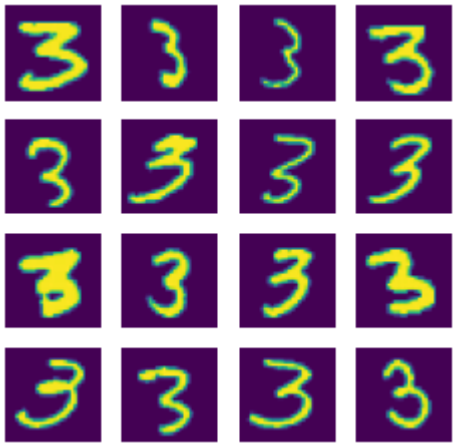
Example of MNIST numbers:



3) We will generate only one number for better quality and performance. So we need to filter data.

```
indices = np.where(y==3)[0]
x = x[indices]
```

Real data examples after filtering:



4) Build generator model. Input data is vector of random normal distributed values with length 128. There are 5 Conv2DTranspose layers in the model. Input shape is (128,) and output shape is (28, 28, 1), like in real data.

Total params: 7,017,729

Trainable params: 7,015,809

Non-trainable params: 1,920

```
noise_size = 128
generator = Sequential([
    Dense(512, input_shape=[noise_size]),
    Reshape([1,1,512]),

    Conv2DTranspose(512, kernel_size=4, use_bias=False),
    BatchNormalization(),
    LeakyReLU(),

    Conv2DTranspose(256, kernel_size=4, use_bias=False),
    BatchNormalization(),
    LeakyReLU(),

    Conv2DTranspose(128, kernel_size=4, strides=2, padding='same', use_bias=False),
    BatchNormalization(),
    LeakyReLU(),

    Conv2DTranspose(64, kernel_size=4, padding='same', use_bias=False),
    LeakyReLU(),
    BatchNormalization(),

    Conv2DTranspose(1, kernel_size=4, strides=2, padding="same", activation='sigmoid'),
])
```

5) Build discriminator model. There are 5 Conv2D layers in the model. Input shape is (28, 28, 1) and output is only one value with activation function sigmoid. It's used to predict if image is real (1) or generated by generator model (0). Metric for this model is accuracy, loss function is binary crossentropy and optimizer is adam.

Total params: 6,952,257

Trainable params: 6,950,337

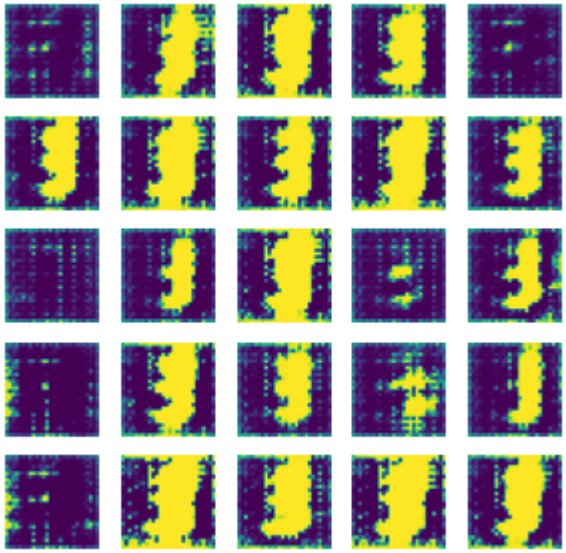
Non-trainable params: 1,920

```
discriminator = Sequential([
    Conv2D(64, kernel_size=4, strides=2, padding="same", input_shape=[28,28,1]),
    BatchNormalization(),
    LeakyReLU(0.2),

    Conv2D(128, kernel_size=4, strides=2, padding="same", use_bias=False),
    BatchNormalization(),
    LeakyReLU(0.2),

    Conv2D(256, kernel_size=4, strides=2, padding="same", use_bias=False),
    BatchNormalization(),
    LeakyReLU(0.2),

    Conv2D(512, kernel_size=4, strides=2, padding="same", use_bias=False),
    BatchNormalization(),
    LeakyReLU(0.2),

    Conv2D(512, kernel_size=4, strides=2, padding="same", use_bias=False),

    Flatten(),
    Dense(1, activation='sigmoid'),
])

opt = tf.keras.optimizers.Adam(learning_rate=2e-4, beta_1=0.5)
discriminator.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
```

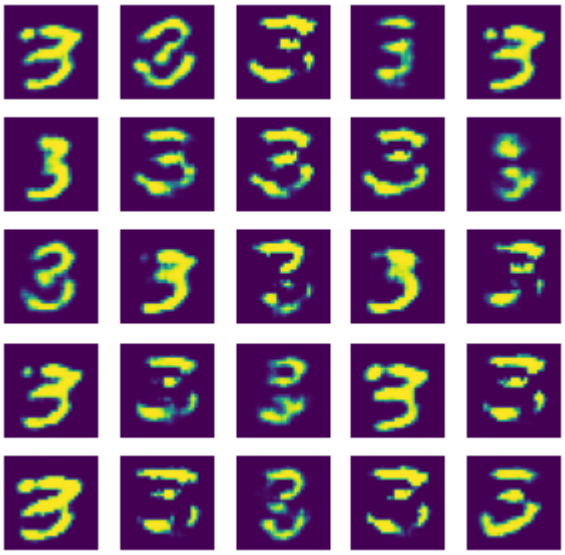
6) For building GAN, I just need to place generator output to discriminator input and check the accuracy of results. So, I used same optimizer, loss function and metric.

```
input_layer = tf.keras.layers.Input(shape=(noise_size, ))
gen_out = generator(input_layer)
disc_out = discriminator(gen_out)

gan = Model(
    input_layer,
    disc_out
)

discriminator.trainable = False
gan.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
```

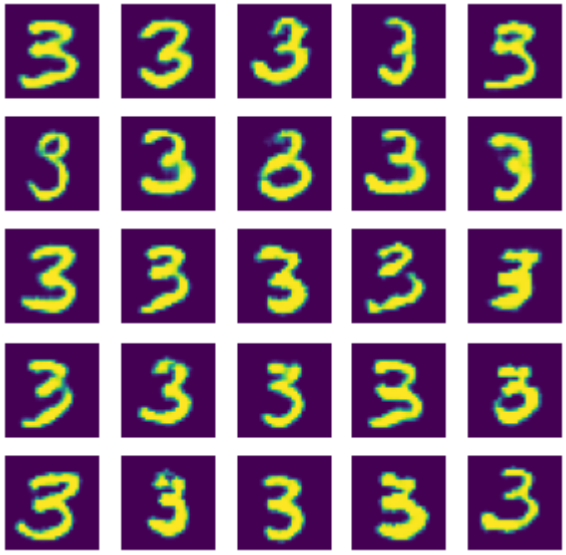
7) To train a model in each of 50 epoch in each iteration (Niterations = Nexamples / batch_size) I get batch of 128 true MNIST examples and add to this batch 128 generated images. After that I train discriminator and then GAN (so and generator) models on this batch. After 20 epochs generated images are looking good.



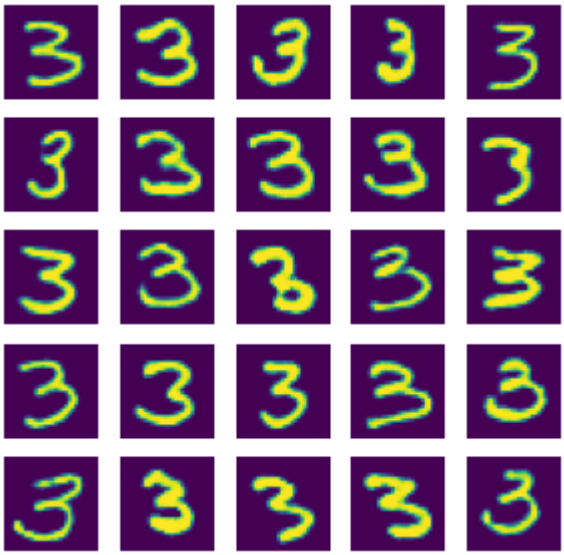
Epoch 2



Epoch 13

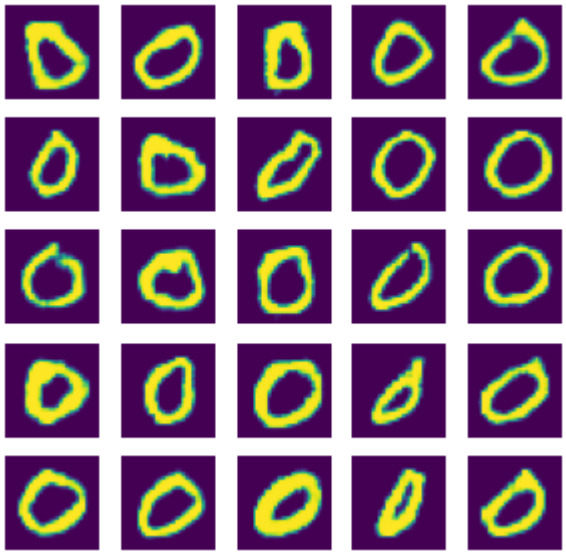


Epoch 20

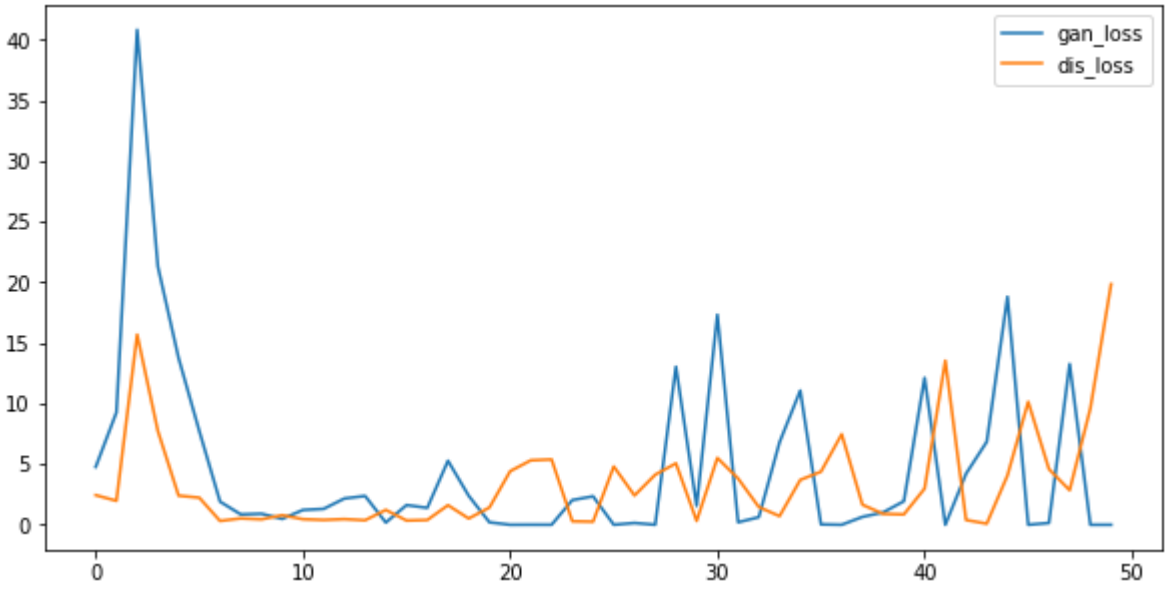


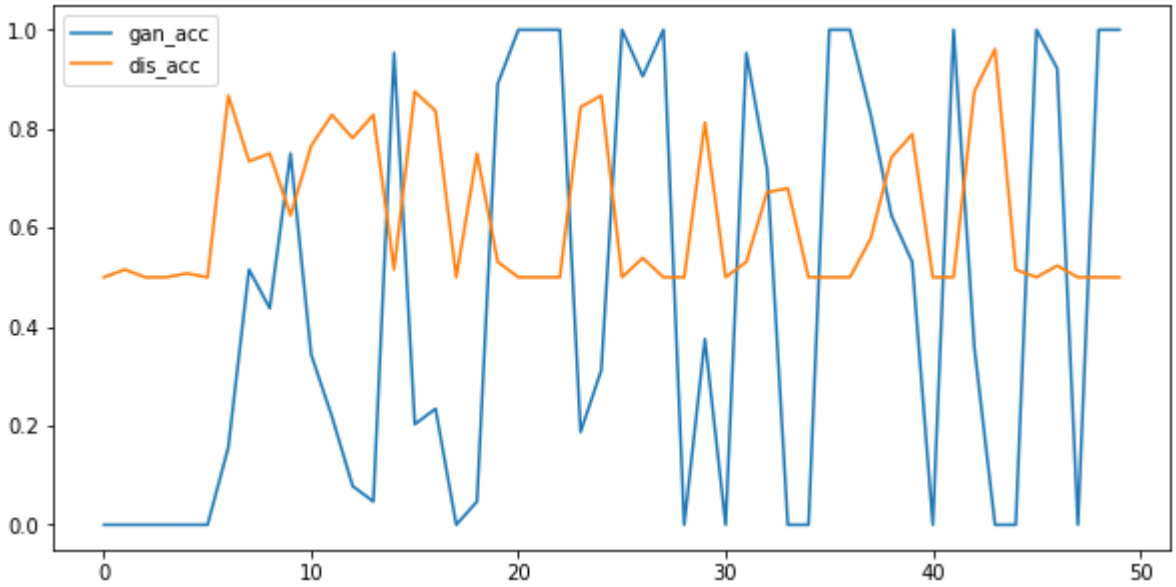
Epoch 42

For zeros I got next images after 42 epochs:



In next images you can see how changes models' accuracy and losses. Discriminator accuracy 0.5 means that it can't recognize fake images from real. You can also notice that the indicators are in antiphase: when the accuracy of the discriminator increases, the accuracy of the generator decreases and vice versa.





Other examples

I also tried to use this DCGAN for generating much complex images, like number nine from SVHN or all MNIST numbers, but was not satisfied with results. In both cases, I got the outlines of the desired numbers, but they are very easy to distinguish from the original. Perhaps the net just needs more epochs to train, but I don't have enough computing power.