Neural Identities

Konrad Werbliński, Wiktor Garbarek
Jakub Kuczkowiak

February 20, 2019

Abstract

Deep Convolutional Generative Adversarial Networks are proven to deliver spectacular results in fake identities generation. In this paper we show how to train some networks on CelebA dataset and use it to generate faces. Moreover the whole operation is approximately reversible, we are able to extract "noise" vectors and mix them together to generate authentic faces similar to what we feed the network with.

Contents

1 Introduction

We begin the following project by describing the motivation and reasons that stand behind our work. We were fascinated by the possibilities of neural networks and we decided to make use of them. There are two main goals that we are to achieve. First of all, we would like to generate artificial face identities and then we would like to be able to mix them with other pictures so that we get the best features of either and therefore can see a single step of evolution. There are numerous reasons we may want to do that. For instance, imagine yourself looking closer to Arnold Schwarzenegger, adding your face a touch of masculinity or predicting how you might look like getting older by similarities to your father and other relatives' faces. It might as well serve us to create fake identities for web services heavily based on the look and attractiveness. [Sha18]

2 Description

To formalize our problem, the neural model is created. We trained the neural network in advance using a large data set of attractive models and profiles with clear and visible faces. Due to time complexity of the solution, we need to compromise on the fact that we used small resolution pictures for the training and perform upsampling on the results to make edges more smooth and dynamic as well as get rid of pixelation.

2.1 Generative Adversarial Network

We were inspired by the article about "Celebrity Face Generation using GANs" using Generative adversarial networks which are considered the hottest topic in deep learning citing [ganb]. GAN consists of two elements: *Generator* and *Discriminator* being in fact reverse neural networks which outputs are put in [0, 1] probability range using sigmoid function.

2.1.1 Discriminator

We can think about both of them as a zero-one game where generator tries to cheat discriminator and discriminator tries to make it impossible to cheat. The main goal of discriminator is to detect bad and fake data while being still consistent with the learning dataset.

2.1.2 Generator

Generator is the one responsible for generating new data and passes them to discriminator in a hope for being qualified as authentic and therefore cheat discriminator. It is a never-ending war between both of them. Discriminator gets better and better in detecting fake images whilst generator increases its capabilities to create world class perfect but fake datasets. Citing [gana], "You can think of a GAN as the combination of a counterfeiter and a cop in a game of cat and mouse, where the counterfeiter is learning to pass false notes, and the cop is learning to detect them. Both are dynamic; i.e. the cop is in training, too (maybe the central bank is flagging bills that slipped through), and each side comes to learn the other's methods in a constant escalation."

2.1.3 Convolutional neural network

It is important to notice that both of them are in fact convolutional neural networks that learn with time. The generator uses random noise and makes an image from it so that there are new generations of dataset each time while discriminator is just a simple classifier that labels whether the image is fake or real. In that sense they are exactly inverse to each other, which means that we can reverse the process. The goal is to minimize the following function:

$$\min_G \max_D V(D,G) = \mathop{\mathbb{E}}_{x \sim p_{data}(x)} [logD(x)] + \mathop{\mathbb{E}}_{y \sim p_y(y)} [log(1 - D(G(y))]$$

The first expected value represents logarithm probability of Discriminator predicting that real-world data is genuine whilst the other represents logarithm probability of Discriminator predicting that data generated by Generator is not authentic. To get the probability in the right range logistic sigmoid function is used:

$$f(x) = \frac{1}{1 + e^{-x}}$$

With GANs it is easy to use the full power of back-propagation however it is also similarly easy to get stuck in a local optima.

2.2 Upsampling

2.2.1 Motivation

Why do we need to perform upsampling? First of all, training the neural network is a very time consuming process and even for a low resolution like 56 x 56 pixels it might take a few days, not saying about increasing the resolution twice. In order to be able to analyze some results we want therefore to work with small pictures and after the neural network produces some output we want to use upsampling to scale them up. It creates a natural problem of describing intermediate pixels. We need algorithms that will try to guess the contents of them analyzing surroundings.

2.2.2 Algorithms

There are known different methods of repairing pictures. They are all based on guessing the contents in order to add smoothness together with sharpness to edges. We don't really care about time and space complexity of upsampling solution because neural networks already take a huge order of running time and so it is most important to focus on accuracy. If we wanted to extend implementation to produce gifs or videos however it would become a very slow and critical operation. One of the most commonly used methods is Fourier interpolation [Fou]. There's also quite an interesting idea presented by Andrea Mazzoleni [Maz]. Some good patent results might be observed using $ImageMagic$ program ([ima]). There are also methods like $Sinc$ or $Lanczos$, which in theory should give the best possible reconstructions but in practice assumptions put by these algorithms fail in practice, both of them being some kind of cubic interpolations.

2.2.3 Edges problem

It is also worth pointing out that we want to use an edge-directed interpolation [edg] since the face should contrast well with background and so it is quite important here to preserve edges. We mention that in 2013, Directional Cubic Convolution Interpolation was well tested on a huge database and passed very well when it comes to edges.

2.2.4 Downsampling

It turns out that not only we come across problems with upsampling but we also want to get rid of downsampling completely. All algorithms under class of interpolation are 'sampling' some determined number of pixels. While scaling the picture down under some level the algorithm won't have those intermediate pixels and might fail, which results in a very huge data loss and gives bad results in practice. There are ideas like box sampling which use huge area around the pixel to find a perfect candidate but still it is not perfect and we really want to avoid doing that.

2.2.5 Small resolution

In reality we should deal with very huge pictures in order to get perfect results. If there is a need to deal with very low resolution (up to 256x256) or relatively small number of colours (between 2 and 256) it should be worth considering hqx algorithm, whose description can be found in [hqx]. Our neural network is predestined to work with real life pictures and therefore we do not handle the implementation of hqx here but we are aware of how to extend it under such assumptions.

2.2.6 Neural networks again

The solution might be creating another neural network responsible for scaling up the images. There are programs implenting it such as $Waifu2x$ and $Reshade$. Also, researchers from Max Planck institute found out a really cool algorithm based on neural networks and known as $EnhanceNet-PAT$ with suprising results. Still, such networks work relatively slow (similarly to the GAN where we avoided using huge resolution images) and there are no formal proofs of how well they behave and therefore we avoid using them.

2.2.7 Solution

In order to get the best possible results with a relatively low time complexity we decided to use *OpenCV* library [cv2a] for Python using cubic interpolation with special filters that gave the best results for given pictures. It is considered as one of the best computer vision implementation and it can be well analyzed since its code is open source and can be found here [cv2b].

3 Implementation

The implementation is done in Python. We use preprocessed (i.e. images were aligned and cropped) CelebA dataset (available at [cel]).

3.1 Preprocessing and dataset loading

After loading whole[1] dataset using pillow library, we apply next preprocessing trick to remove most of non-face parts of photos. Arguments delta_j, delta_i, rotation and face_size are optional in case of trying to manually load desired photo to align it. (Default values are 108 for face_size and 0 for delta_j, delta_i and rotation).

```
def celeba_preprocessing(image, width, height, **kwargs):
    face_width = face_height = kwargs.get('face_size', 108)
    j = (image.size[0] - face_width) // 2 + kwargs.get('delta_j', 0)
    i = (image.size[1] - face_height) // 2 + kwargs.get('delta_i', 0)
    return image.rotate(kwargs.get('rotation', 0)) \
            .crop([j, i, j + face_width, i + face_height]) \
            .resize([width, height], Image.BILINEAR)
```

Every photo is normalized from integer values $[0, 255]$ to $[-0.5, 0.5]$ by following code, where preprocessing_function in our case is function above

```
def load(image_files, mode,
         preprocessing_function=lambda x, **kw: x,
         **preprocessing_kwargs):
    data_batch = np.array(
        [get_image(sample_file, mode,
                   preprocessing_function,
                   **preprocessing_kwargs)
         for sample_file in image_files]).astype(np.float32)

    if len(data_batch.shape) < 4:
        data_batch = data_batch.reshape(data_batch.shape + (1,))

    return data_batch / 255 - 0.5
```

3.2 Network architecture

Below is our DCGANs architecture

```
    Generator:
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_17 (Dense)             (None, 50176)             20120576
_________________________________________________________________
batch_normalization_33 (Batc (None, 50176)             200704
_________________________________________________________________
leaky_re_lu_65 (LeakyReLU)   (None, 50176)             0
_________________________________________________________________
reshape_9 (Reshape)          (None, 14, 14, 256)       0
```

[1] "lazy" loading while getting batches proven to be less effective and significantly lengthened training time

```
_________________________________________________________________
dropout_41 (Dropout)         (None, 14, 14, 256)       0
_________________________________________________________________
up_sampling2d_17 (UpSampling (None, 28, 28, 256)       0
_________________________________________________________________
conv2d_transpose_33 (Conv2DT (None, 28, 28, 128)       819328
_________________________________________________________________
batch_normalization_34 (Batc (None, 28, 28, 128)       512
_________________________________________________________________
leaky_re_lu_66 (LeakyReLU)   (None, 28, 28, 128)       0
_________________________________________________________________
up_sampling2d_18 (UpSampling (None, 56, 56, 128)       0
_________________________________________________________________
conv2d_transpose_34 (Conv2DT (None, 56, 56, 64)        204864
_________________________________________________________________
batch_normalization_35 (Batc (None, 56, 56, 64)        256
_________________________________________________________________
leaky_re_lu_67 (LeakyReLU)   (None, 56, 56, 64)        0
_________________________________________________________________
conv2d_transpose_35 (Conv2DT (None, 56, 56, 32)        51232
_________________________________________________________________
batch_normalization_36 (Batc (None, 56, 56, 32)        128
_________________________________________________________________
leaky_re_lu_68 (LeakyReLU)   (None, 56, 56, 32)        0
_________________________________________________________________
conv2d_transpose_36 (Conv2DT (None, 56, 56, 3)         2403
_________________________________________________________________
activation_17 (Activation)   (None, 56, 56, 3)         0
=================================================================
Total params: 21,400,003
Trainable params: 21,299,203
Non-trainable params: 100,800
_________________________________________________________________

Discriminator:
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_33 (Conv2D)           (None, 28, 28, 64)        4864
_________________________________________________________________
leaky_re_lu_69 (LeakyReLU)   (None, 28, 28, 64)        0
_________________________________________________________________
dropout_42 (Dropout)         (None, 28, 28, 64)        0
_________________________________________________________________
conv2d_34 (Conv2D)           (None, 14, 14, 128)       204928
_________________________________________________________________
leaky_re_lu_70 (LeakyReLU)   (None, 14, 14, 128)       0
_________________________________________________________________
dropout_43 (Dropout)         (None, 14, 14, 128)       0
_________________________________________________________________
conv2d_35 (Conv2D)           (None, 7, 7, 256)         819456
_________________________________________________________________
leaky_re_lu_71 (LeakyReLU)   (None, 7, 7, 256)         0
_________________________________________________________________
dropout_44 (Dropout)         (None, 7, 7, 256)         0
_________________________________________________________________
conv2d_36 (Conv2D)           (None, 7, 7, 1024)        6554624
_________________________________________________________________
```

```
leaky_re_lu_72 (LeakyReLU)   (None, 7, 7, 1024)        0
_________________________________________________________________
dropout_45 (Dropout)         (None, 7, 7, 1024)        0
_________________________________________________________________
flatten_9 (Flatten)          (None, 50176)             0
_________________________________________________________________
dense_18 (Dense)             (None, 1)                 50177
_________________________________________________________________
activation_18 (Activation)   (None, 1)                 0
=================================================================
Total params: 7,634,049
Trainable params: 7,634,049
Non-trainable params: 0
_________________________________________________________________

DM:
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
discriminator (Sequential)   (None, 1)                 7634049
=================================================================
Total params: 7,634,049
Trainable params: 7,634,049
Non-trainable params: 0
_________________________________________________________________

AM:
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
generator (Sequential)       (None, 56, 56, 3)         21400003
_________________________________________________________________
discriminator (Sequential)   (None, 1)                 7634049
=================================================================
Total params: 29,034,052
Trainable params: 21,299,203
Non-trainable params: 7,734,849
_________________________________________________________________
```

3.3 Image reversion

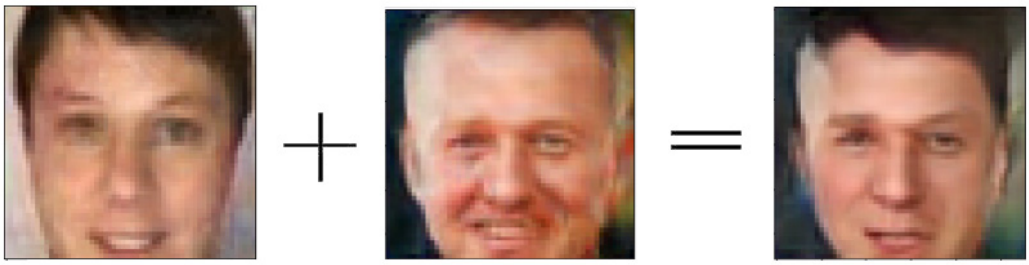
Getting noise vector from image is done by simple loss minimization

```
def gan_reverse(i, steps = 50):
    z = np.random.normal(0, 0.5, size = [1, input_shape])
    x = K.placeholder()
    loss = K.sum(K.square(i - Gen.outputs[0]))
    grad = K.gradients(loss, Gen.inputs[0])[0]
    update_fn = K.function(Gen.inputs, [loss, grad])

    def eval_loss_and_grad(z, j):
        l, g = update_fn([z.reshape(1, input_shape), j])
        return l.astype('float64'), g.astype('float64')

    for step in range(steps):
        j = Gen.predict(z, steps=1)
        f = lambda x: eval_loss_and_grad(x, j)
        z, min_val, _ = fmin_l_bfgs_b(f, z, maxfun=20)
        z = z.reshape(1, input_shape)
```

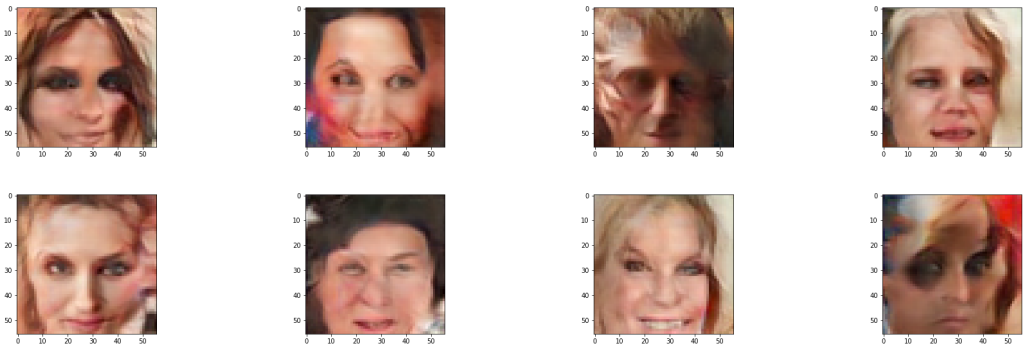
```
        if step % 5 == 0:
            print("Step:", step)
    return z
```

4 Testing and Results

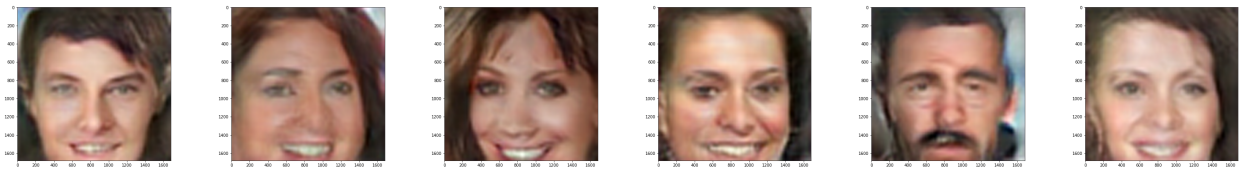
As mentioned before we trained our neural networks on images of size 56x56. It turned out to be difficult to get a higher resolution in a reasonable time using the machines we were in possession of. There were also difficulties in converging of GANs for resolution of size as small as 112x112. We therefore present our results in a small, yet quite clear resolution.
In the first sample we can one of the authors, Konrad Werbliński, mixed together with Arnold Schwarzenegger.

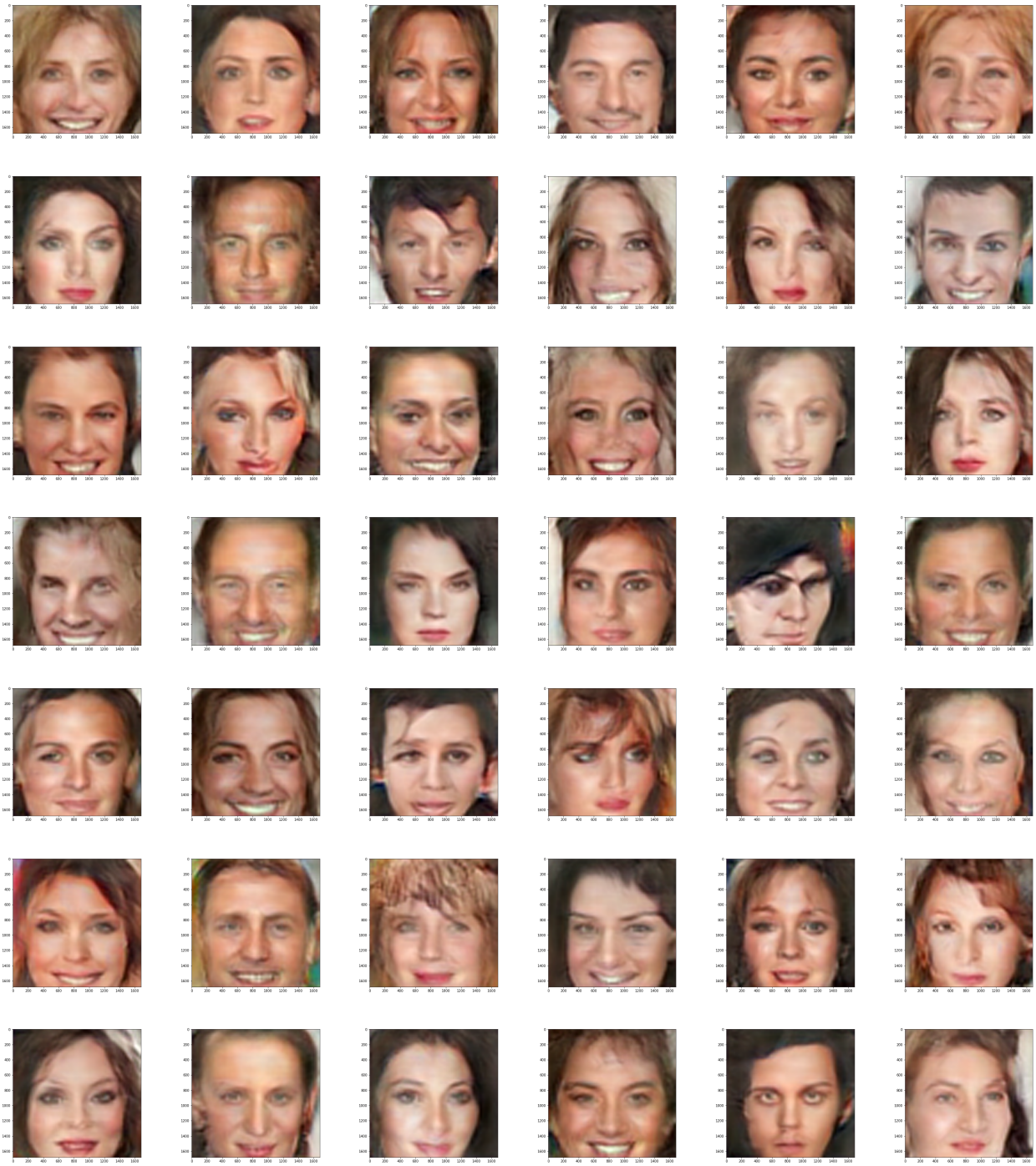


Below we might see some surprising and unexpected results of our network which reminds of 'The Walking Dead' series. We are sure one can find a great usage of them.



Finally, we present some of the generated identities by our network:





5 Summary

The results seem very promising. Two faces can be successfully mixed together and GAN can also produce great fake identities. What next? It would be worth trying it on another dataset, especially one with very high resolution. It could give interesting results for objects other than people's faces (there is an interesting work that transforms horses to zebras and the other way around [cyc]). There are more interesting ways to perform mutation. We used arithmetic average and it should be also possible to clarify which elements of vector are responsible for particular features of face. Remains question if features are somehow related to each other but it is a topic for completely different story.

References

[cel] CelebA source.

[cv2a] CV2.

[cv2b] CV2 Implementation.

[cyc] GAN with zebras and horses.

[edg] edgedirectore. Edge-Directed interpolation.

[Fou] Fourier-based interpolation bias prediction in digital image correlation.

[gana] GAN description.

[ganb] GAN source.

[hqx] hqx.

[ima] Image Magic.

[Maz] Andrea Mazzoleni. Scale2x.

[Sha18] Shubham Sharma. Celebrity Face Generation using GANs (Tensorflow Implementation). August 2018.