Original Face Generation using Adversarial Generative Networks

Richard Bell, Eric Ho, Alexander Khoury, Shengyao Guo
University of California, San Diego
9500 Gilman Dr, La Jolla, CA 92093
rcbell@eng.ucsd.edu, erho@ucsd.edu,
ackhoury@eng.ucsd.edu, s8guo@ucsd.edu

Abstract

This paper explores the use of generative adversarial networks (GANs) to generate human like faces that are similar to existing human faces. A number of different similarity metrics between images will be used and the resulting images presented. A discussion on how GANs work and why they are useful will be presented, as well as a comparison between GAN generated faces and a more traditional generation technique.

1. Introduction

In this project we wanted to know if a GAN could be used to generate realistic looking faces that are similar to a target face. To answer this question we implemented a DCGAN using TensorFlow and the CelebA dataset. We experimented with different DCGAN configurations, and parameters. We also used several different metrics to quantify the notion of similarity between faces.

Generating realistic faces is desirable for industries that rely on faces as part of their business model. For example, the advertising industry pays male and female models large amounts of money to endorse products such as food, clothing, cars, sports and movies. Using computer generated images would save the industry and subsequently consumers money. Another area that would benefit from image generation is image reconstruction. If an image has been damaged such that portions of pixels are missing, or were blocked by an obstruction, a generative model such as the one presented here can be used to fill in these missing pixels. For more information on the practical usefulness of GANs see [3].

An example of faces generated by a GAN made by NVIDIA [5] is shown in Figure 1. These faces are good enough to fool a large majority of people into thinking they are real. The training set used to create this output is called the Large-scale CelebFaces Attributes (CelebA) dataset [2] and is the same dataset that will be used in this paper to train our GANs. This dataset consists of 202,599 celebrity images each with 40 labeled features. The faces in each image have been (mostly) centered and are facing the camera.

Figure 1: Original faces created by NVIDIA using a GAN trained with the CelebA dataset.

2. Methods

2.1. Traditional Approach

The special effects and gaming industry have used traditional techniques to blend and morph faces for decades. These techniques typically do not involve machine learning nor do they depend on any statistical properties of faces in general. A survey of traditional face morphing techniques is given in [6]. The technique we chose to use for comparison purposes begins with two images, I and J. The goal is to create a third image, M, that is an affine transformation of the original two

$$M(x, y) = (1 - \alpha)I(x, y) + \alpha J(x, y)\,, \qquad (1)$$

where x and y are the coordinates of a pixel and $0 \leq \alpha \leq 1$ is the parameter that controls the proportion that each input image contributes to the output. However, applying this transformation directly to two input images without some

Figure 2: The result (middle) of applying (1) to the left and right images.

Figure 3: The result (middle) of applying (1) to the left and right images after preprocessing the input images so that important landmarks and regions are aligned.

preprocessing will produce unsettling results as shown in Figure 2.

To improve the results of this technique, point correspondences should be identified on each input image. This amounts to placing markers on important facial landmarks such as the nose, lips, eyes, eyebrows, jawline and ears. This can be done manually or with some automated facial recognition technique. The same number of points should be placed on each image. Next each point on each image is connected with other points so as to produce a partitioning of the image into triangles, a process called Delaunay Triangulation [7]. Using (1), create the output image landmark points from the landmark points identified on the input images and form the same triangulation on the output image. We now have a one-to-one correspondence between input triangles and output triangles. Now pick a triangle from the first input image and the corresponding triangle from the output image and compute the affine transformation needed to map the three corners of the input triangle to the output. Repeat this process for the same triangle in the second input image. Use these transformations to convert all pixels from the two input images to the output image. Finally apply (1) to these warped triangles to produce the final result shown in Figure 3.

While the preprocessing greatly improved the output image, it is evident that the face is a mixture of the inputs and not a truly original face. By using GANs, more realistic and original faces can be produced.

2.2. GANs

Generative adversarial networks were introduced by Goodfellow *et al*. [4] in 2014 as a generative unsupervised learning technique that can be used to either estimate probability densities directly, produce samples from a distribution, or both. In this paper we will use a GAN to generate sample images defined by unlabeled training data in the form of the CelebA data set. Though it was mentioned earlier that this dataset is labeled, the GAN itself does not use these labels. We will use this labels to create different In essence what the GAN does is produce data that is similarly distributed to the training data.

A GAN is composed of a generator and a discriminator. The generator produces data samples that are fed to the output of the GAN as well as the input of the discriminator. The discriminator determines whether a generator output sample is from the training distribution or the generators output data distribution, data distribution for short. Thus the generator and discriminator can be thought of as players in a two player game. The generator wins the game by fooling the discriminator and the discriminator wins by picking the correct distribution. The competition between generator and discriminator drives the data distribution to be indistinguishable from the training distribution.

It is easiest to implement a GAN using multilayer perceptrons for both the generator and discriminator due to the ease of using backpropagation to train them. Let G be the function implemented by the generator multilayer perceptron and D the discriminator function. The function G takes a vector noise input, $\boldsymbol{z} \in \mathbb{R}^p$ with prior distribution $p_Z(\boldsymbol{z})$ and produces output vectors $\boldsymbol{x} \in \mathbb{R}^{nm}$ distributed as $p_g(\boldsymbol{x})$, where the output image has $n \times m$ pixels. Thus the generator implements

$$\boldsymbol{x} = G(\boldsymbol{z}; \theta_g)\,, \tag{2}$$

where θ_g are the weights that parameterize the generator function.

The discriminator function D takes the generator output vectors, $\boldsymbol{x}$, as input and produces a scalar output, $y \in [0, 1]$, representing the probability that $\boldsymbol{x}$ came from the generator distribution or the training distribution

$$y = D(\boldsymbol{x}; \theta_d)\,. \tag{3}$$

The discriminator is trained to maximize the probability of assigning the correct label to generator samples and training samples. The generator is trained to minimize the objective $J_G = \log(1 - D(G(\boldsymbol{z})))$. This leads to the following optimization problem presented in [4]

$$\min_G \max_D E_{\boldsymbol{x}}[\log D(\boldsymbol{x})] + E_{\boldsymbol{z}}[\log(1 - D(G(\boldsymbol{z})))]\,. \tag{4}$$

It can be shown that when the parameterizations θ_g and θ_d are not a limiting factor that the solution functions G and D to (4) will produce data with density $p_g = p_{\text{train}}$. The backpropagation is typically implemented using a mini-batch stochastic gradient descent, however, any number of backpropagation schemes can be applied.

2.3. DCGANs

When using GANs for image generation as described here, there are some problems that can be encountered as pointed out in [9]. The images the GAN produces can look wobbly or blurry and the GAN can become unstable during training. Thus a deep convolutional GAN (DCGAN) structure was proposed and tested and shown to produce better results. This is the implementation chosen in this paper.

The general structure of the DCGAN is shown in Figure 4. An input noise vector is fed into the generator which traverses through four transpose convolutional layers and one linear layer to produce an output image. Batch normalization with ReLU activation functions are also used with $\tanh$ at the output layer. The output image is then fed into the discriminator which traverses through four convolutional layers and one linear layer to produce a probability of being a real image or a generated one. Batch normalization and leaky ReLU layers are also used.

2.4. Similarity Metrics

It is desirable to quantitatively define the similarity between two images instead of a subjective comparison. Malik and Baharudin [8] define several metrics that can be used for this purpose. We briefly present some of the metrics used in this paper now.

For each of the distance metrics described next, define F_i as the value of the i^{th} feature out of n for the first image and Q_i as the i^{th} feature out of n for the second image, where $1 \leq i \leq n$. The distance between the two images will be given by Δd.

2.4.1 Euclidean Distance

$$\Delta d = \sqrt{\sum_{i=1}^{n} \left(|F_i - Q_i|\right)^2} \tag{5}$$

2.4.2 City Block Distance

$$\Delta d = \sum_{i=1}^{n} |F_i - Q_i| \tag{6}$$

2.4.3 Sum of Squares of Absolute Difference

$$\Delta d = \sum_{i=1}^{n} \left(|F_i| - |Q_i|\right)^2 \tag{7}$$

2.4.4 Canberra Distance

$$\Delta d = \sum_{i=1}^{n} \frac{|F_i - Q_i}{|F_i| + |Q_i|} \tag{8}$$

2.4.5 Maximum Value Distance

$$\Delta d = \max\{|F_i - Q_i|, \ldots, |F_n - Q_n|\} \tag{9}$$

2.4.6 Sum of Absolute Difference

$$\Delta d = \sum_{i=1}^{n} \left(|F_i| - |Q_i|\right) \tag{10}$$

2.4.7 Minkowski Distance

$$\Delta d = \left[\sum_{i=1}^{n} \left(|F_i - Q_i|\right)^p\right]^{\frac{1}{p}} \tag{11}$$

2.5. Feature Extraction

Before we can use the similarity metric, however, it is necessary to extract useful features for each image. Two different methods will be used to achieve this: principal component analysis (PCA) and discrete cosine transform (DCT) with statistical texture features.

2.5.1 PCA

In the first method of feature extraction, the principal components (PCs) of each image were used as features. First, the data is centered so that the average of the images is zero. Next, the data is normalized by dividing each feature by the standard deviation so that the PCs are scale invariant. The next step is to apply singular value decomposition (SVD) to find the thirty-two largest eigenvalues so we can project the images to the associated eigenvectors. In doing so the data has been reduced in dimension to that of thirty-two features.

2.5.2 DCT and Texture Features

The second method used to extract features relies on the DCT. The DCT of an image patch is computed and used to calculate the statistical texture features, namely mean, standard deviation, skewness, kurtosis, energy and smoothness. These features are useful for content based image

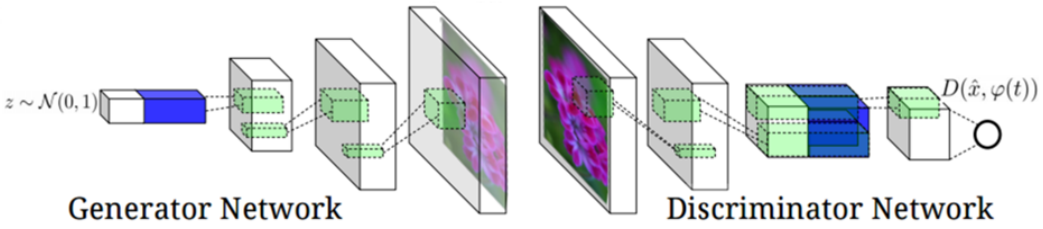


Figure 4: The general structure of the DCGAN used in this paper.

retrieval systems as shown in [8]. The DCT is applied to non-overlapping 8x8 blocks of an image with all the DC coefficients stored to a vector v_{DC} and the first three AC coefficients stored to vectors v_{AC1}, v_{AC2} and v_{AC3}. Next, a histogram is made for DC and AC coefficients using thirty-two bins and used to calculate the probability distribution of bin b for each of the four histograms

$$P(b) = \frac{H(b)}{n}, \tag{12}$$

where $H(b)$ is the histogram of a coefficient and n is the total number of 8x8 blocks of the image. Using the probability distributions $P(b)$, we can now calculate the texture features.

The mean represents the brightness of the image which is the average of the intensity values for all bins of each coefficient: $\mu = \sum_{b=1}^{32} bP(b)$. The standard deviation encodes the amount of contrast in the image: $\sigma = \sqrt{\sum_{b=1}^{32}(b-\mu)^2 P(b)}$. The skewness shows how the distribution of intensity values are distorted: skew $= \frac{1}{(std)^3}\sum_{b=1}^{32}(b-\mu)^3 P(b)$. The kurtosis measures the peak of the distribution: kurt $= \frac{1}{\sigma^4}\sum_{b=1}^{32}(b-\mu)^4 P(b)$. Energy is indicative of the uniformity of the distribution throughout the bins of the histograms: energy $= \sum_{b=1}^{32}(P(b))^2$. Lastly, the smoothness measures the surface properties of the image: smooth $= 1 - \frac{1}{1+\sigma^2}$. Once the features are computed, they are put into an array creating a feature vector. For example, for the DC coefficient $fv_{DC} = [\mu, \sigma, \text{skew}, \text{kurt}, \text{energy}, \text{smooth}]$. The final feature vector for an image is an array with each coefficient feature vector stacked together $FV = [fv_{DC}, fv_{AC1}, fv_{AC2}, fv_{AC3}]$.

3. Experiments

3.1. Insight

It was important to verify that we would be able apply the DCGAN to our problem. Thus, at first, we used a previously working implementation of a DCGAN, and were able to successfully generate realistic looking faces.

Figure 5: Female and Male outputs from carpedm20's DCGAN [1] using celebA

Using carpedm20's DCGAN [1], we were able to get female faces that looked quite good; however, the male faces still required more improvement. Using this as a proof of concept, we were able to design and implement our own DCGAN implementation, allowing for more flexibility in the design, as well as a focused solution towards our problem. Upon testing our implementation, we attained results we deem to look more promising than that which was shown in Fig. 5

Figure 6: Female and Male outputs from our DCGAN using celebA

In our implementation, both the generated female and male faces look much more realistic and varied in their features. From here we aimed to determine the optimal parameters of our model.

3.2. Parameter Tuning

There are many tunable hyperparameters of main importance in the implementation of the DCGAN. Among them, we decided to analyze:

1. Learning rate
2. Optimizer method
3. Number of convolutional layers
4. Batch size

Proper choices for these values is very important as a wrong choice could lead to poor results and instability in the network. It also can lead to either the discriminator or generator overpowering the other, which hinders the networks ability to learn; it must remain a *fair* two player game.

In order to investigate the effect of these parameters on our results, we swept each with a set of three choices, and compared performances in generator and discriminator loss over 100 epochs as shown in Figure 7.

1. **Learning Rate:** Learning rate was swept over three values, 0.02, 0.002, and 0.0002. All three resulted in similar convergence, with final losses around 1.4 (d_loss) and 0.6 (g_loss). None of the choices seemed to have a big effect on performance of the network in the long term, however the early epochs differ slightly. Using learning rates of 0.02 or 0.002 seemed to result in a more steady decline. The learning rate of 0.002 resulted in the smallest error for both the generator and discriminator, thus it was the value chosen for future runs.

2. **Optimizer:** Three different optimizers were implemented: Adam, AdaGrad, and Momentum. As easily seen from the loss plots, the only successful optimizer was the Adam optimizer. As mentioned before, the discriminator and generator must remain evenly matched; their even competition drives the learning of the network. As we can see in the case of the AdaGrad Optimizer, the discriminator quickly dominated over the generator, which never allowed a chance for the generator to learn. We can see the generator loss increasing, while the discriminator loss nears 0. In the case of the Momentum Optimizer, a similar story occurred, where the discriminator quickly overpowered the generator. However at around epoch 60, the generator finally was able to improve itself, spiking the discriminator loss, as it needs to successfully classify the better generator images. Towards the later iterations we can see that both networks are learning, unlike the early iterations. Clearly, the Adam optimizer was chosen for future training.

3. **Number of Convolutional Layers:** The three layer sizes used tests three different levels of complexities for the discriminator and generator. We ran with 3, 4, and 5 layer discriminators/generators, and changing this did not effect the losses too much; however with the increase in complexity was an increase in time to train. Looking at the output images, it was clear that a layer size of 4 was the best.

4. **Batch Size** Three different batch sizes were experimented with, 256, 512, and 1024. Batch sizes of 256 and 512 preformed similarly, while a batch size of 1024 presented with some instability in the early iterations. All three in the later iterations converged to similar results. Either 256 or 512 would have worked similarly, so 256 was chosen.

Table 1: Optimal hyperparamters

Learning Rate	0.002
Optimizer	Adam
Num Layers	4
Batch Size	256

From these results, an optimal model was chosen with the parameters given in Table 1.

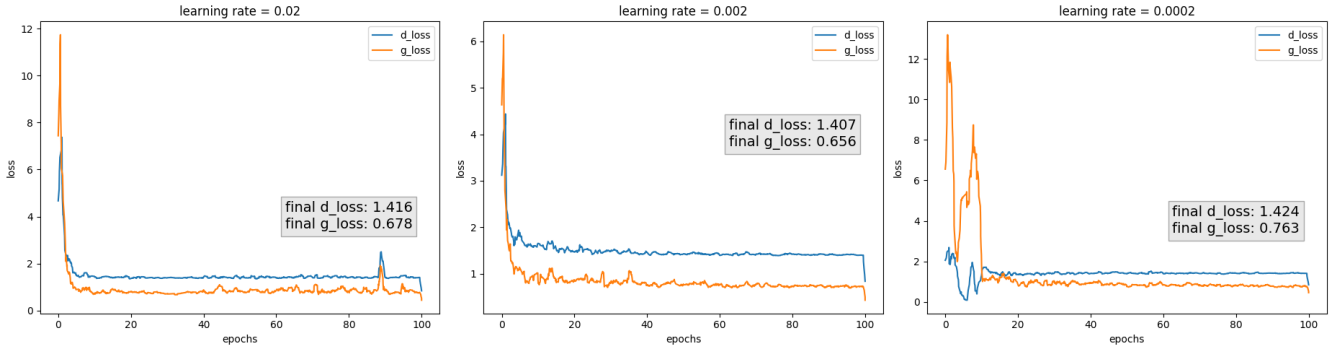
3.3. Tuned Model Results

With a tuned model we were able to generate higher resolution images, (64x64x3 (previous results) $\rightarrow$ 80x80x3), as well as more realistic looking faces. Each of the models – male and female – were trained for 12 hours on a Nvidia GTX 1080Ti. Some of the generated samples are shown in Figures 10 and 11.

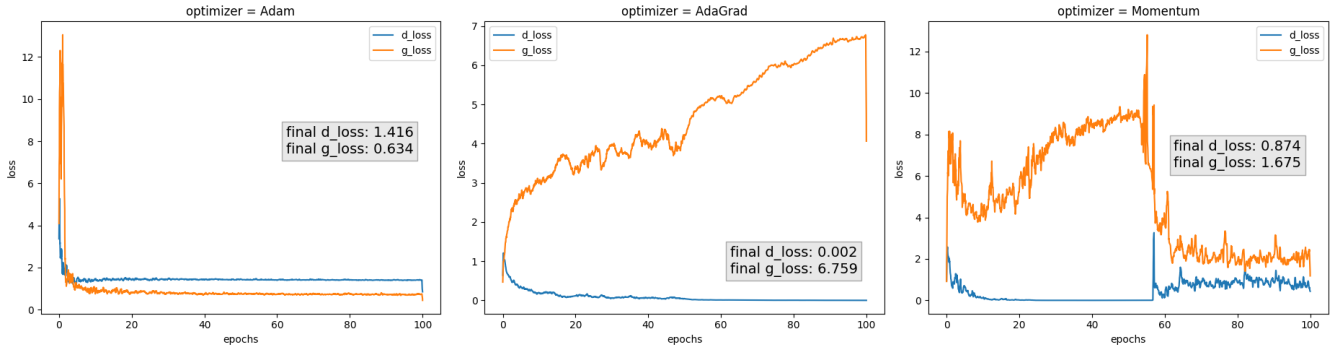
3.4. Visual Turing Test

One the ways we can evaluate the performance of our DCGAN is to create a visual Turing test. Using Google forms, we created a quiz that would present the taker with a real or generated face, and they would have to decide which it is. For the real faces we downsampled images from the training set to match the output resolution of the generator. There were an even amount of generated images as there was real images. Thus, a perfect score for our generator would be 50% as the taker is not able to distinguish the difference between a real and fake image. This became the baseline for performance of our DCGAN.

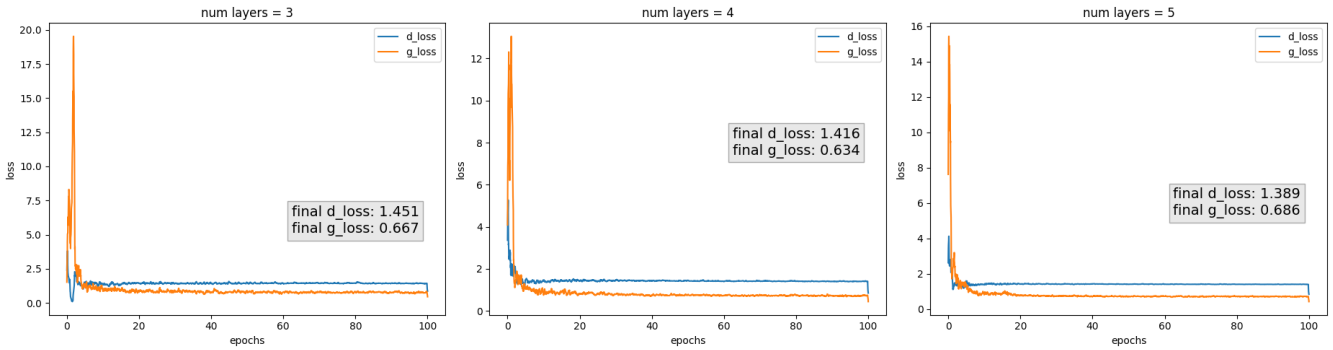
This quiz received a mean score of 74.9%, with a standard deviation of 11.2%. Upon examining the results, there



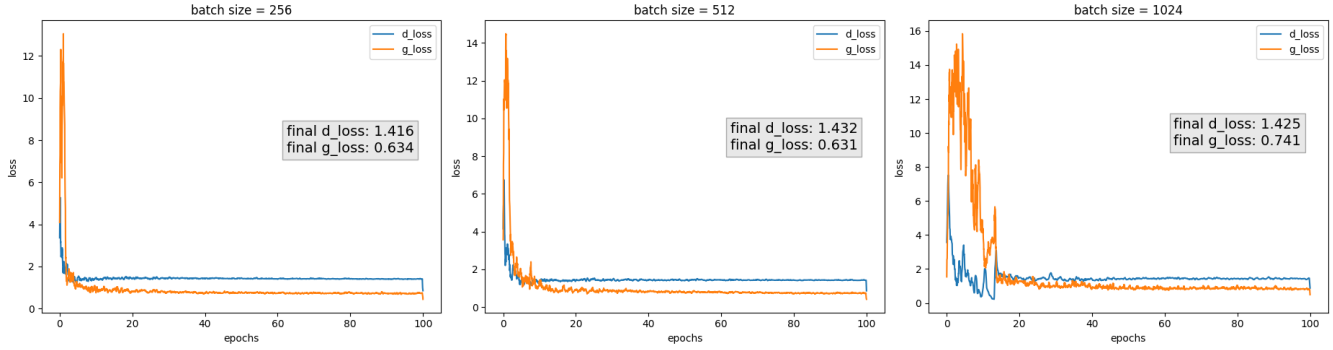
(a) Sweeping learning rate = [.02, .002, .0002]



(b) Sweeping optimizer method = [Adam, AdaGrad, Momentum]



(c) Sweeping number of convolutional layers = [3, 4, 5]



(d) Sweeping batch size = [256, 512, 1024]

Figure 7: Plots of the generator and discriminator loss over 100 epochs for different hyperparameter values.

Figure 8: Male faces generated by our DCGAN

Figure 9: Female faces generated by our DCGAN

were a few of the generated images that achieved/surpassed the 50% baseline, where the quiz takers on average were not able to classify correctly with distinction, shown in Figure 11. The image on the far left was classified as a real image from 59% of participants, the middle with 50%, and the right with 41%.

3.5. Doppelganger Creation

With the fine-tuned model we achieved in the previous section, we generated a face that is similar to the input face with two approaches. In the first approach, we manipulate the input and modify the DCGAN structure to drive the generator producing a desired face. In the second approach, we use the similarity metric to find the closest face in a set of generated faces given the input face.

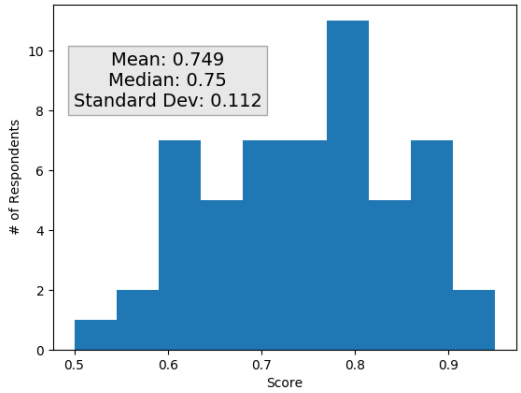


Figure 10: Histogram of Scores from the visual Turing test

Figure 11: Generated examples that received near 50/50 classification

3.5.1 Approach 1

We created the human-like image by replacing the multinomial Gaussian noise input of the DCGAN with a down-sampled version of the target face. Then, we add to the generator loss function an additional term of L1 difference between the generated face and the down-sampled version of the target face. The intuition behind this is to drive the generator output closer towards the target face.

We use the L1 norm for its ability to preserve outliers while driving most of the pixel values towards the target. The L2 norm will penalize the outliers much harder compared to L1 norm, which essentially over fits the pixel values as seen in Figure 12. Visually, the image generated with the L2 norm is incapable of preserving edges and details, which causes the image to look blurry. By comparison, the outlines of all facial landmarks are clear and visible using the L1 norm. Figure 13 illustrates the DCGAN generated face compared to the original face.

3.5.2 Approach 2

In this approach, we found the closest image to the input image using the similarity metric. We first used our DCGAN to generate 3,000 images of faces and we extract the features of these images and the input image using both methods of feature extraction as mentioned in the Methods sec-

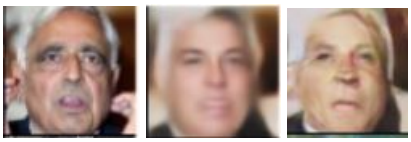
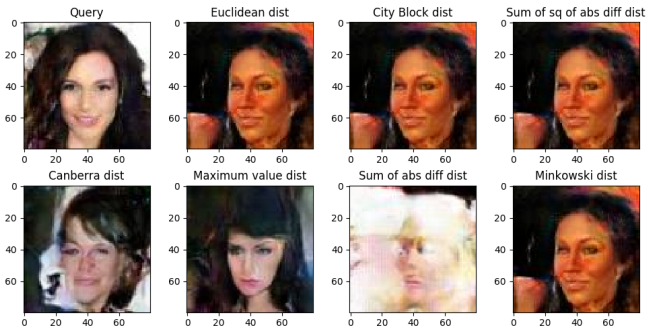


Figure 12: Comparison among original face, L2, and L1

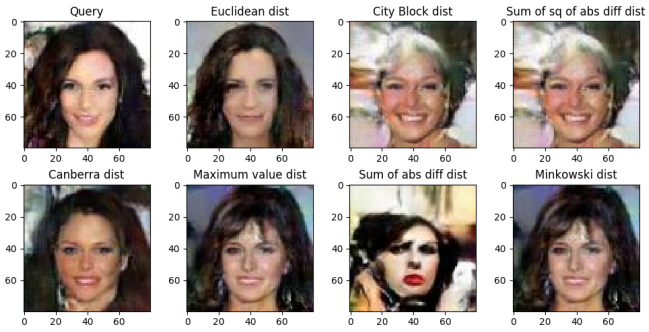
Figure 13: Images of downsampled, interpolated, generated, and original faces.

tion. Next, we calculated the different distances from the input image to each image in the image set.

From Figure 14, we can see that feature extraction using PCA produce better results than using DCT and texture features. Regarding the similarity metric, the best three distances is Euclidean, Canberra, and maximum value distance. On the other hand, the sum of absolute difference distance performs the worst for both PCA and DCT.



(a) Closest faces using DCT and texture features



(b) Closest faces using PCA

Figure 14: Plots of the closest face of an input image.

3.5.3 Comparison

The result generated by each approach has its own merits and drawbacks. Approach 1 is able to generate faces which highly resemble the target faces, however missing out on lots of fine details of facial landmarks. Probable causes include the lack of randomness introduced in the DCGAN, and a linear combination of sigmoid and L1 norm in the loss function is too simple to capture all the features. These factors limit the capability of generalizing good faces. Approach 2 does a much better job to generate general faces, but have a hard time searching for faces that closely match the target face because there are no driving forces that guide the output to resemble the target face. Future work will be searching for a reasonable middle ground between the two approaches and to find the best trade-off between details and similarity.

4. Conclusion

In the end, we have shown that a DCGAN can be used to generate realistic faces. We also demonstrated that these faces can be used to ”create” a face similar to a target face. We also explored using a DCGAN to generate the lookalike face from a downsampled input.

Using the GAN guidelines from Goodfellow’s paper, we created our own DCGAN and found the optimal hyperparameters via parameter sweeps. In order to create lookalike faces, we sampled the trained model for faces, and used various distance measures to find a face in the generated set. For this purpose, the Euclidean distance metric was found to be the best.

The generated faces were not as good as NVIDIA’s due to a lack of computational power, and a simpler GAN structure. They were able to train their GAN for multiple days using a cluster of state of the art hardware to achieve the quality. Also they used a Progressively Growing GAN model, rather than the DCGAN model. However, even with the DCGAN implementation, we were able to create some samples that were able to fool humans.

Our methods for doppelganger creation can be improved significantly, as we suspect that using a Super Resolution GAN structure instead would allow us more control over the faces that are generated.

References

[1] Dcgan-tensorflow. https://github.com/carpedm20/DCGAN-tensorflow. Accessed: 2018-03-19.

[2] Large-scale CelebFaces Attributes dataset (CelebA). http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html. Accessed: 2018-03-19.

[3] I. J. Goodfellow. NIPS 2016 Tutorial: Generative Adversarial Networks. *CoRR*, abs/1701.00160, 2017.

[4] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative Adversarial Networks. *ArXiv e-prints*, June 2014.

[5] T. Karras, T. Aila, S. Laine, and J. Lehtinen. Progressive growing of gans for improved quality, stability, and variation. *CoRR*, abs/1710.10196, 2017.

[6] D. R. Kasat, S. Jain, and V. M. Thakare. Article: A survey of face morphing techniques. *IJCA Special Issue on Recent Trends in Information Security*, RTINFOSEC:14–18, February 2014. Full text available.

[7] D. T. Lee and B. J. Schachter. Two algorithms for constructing a delaunay triangulation. *International Journal of Parallel Programming*, 9(3):219–242, 1980.

[8] F. Malik and B. Baharudin. Analysis of distance metrics in content-based image retrieval using statistical quantized histogram texture features in the dct domain. *Journal of King Saud University - Computer and Information Sciences*, 25:207–218, July 2013.

[9] A. Radford, L. Metz, and S. Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *CoRR*, abs/1511.06434, 2015.