1. Task description

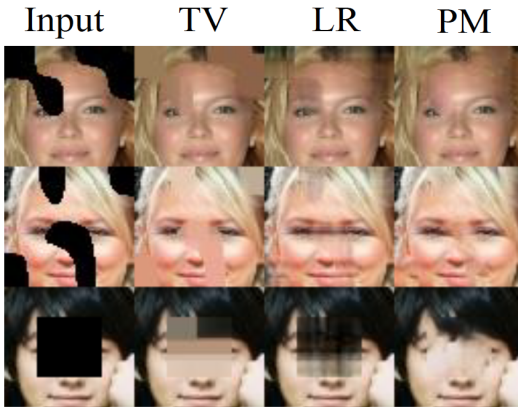
The task of face completion is also well known as Semantic inpainting, which is referred to the task of inferring arbitrary large missing regions in images based on image semantics[1]. The task becomes difficult if the missing area from the picture is large or if the images itself is quite complex. In case of face completion, the complexity increases as it becomes difficult to identify the best suited facial structure.

As part of Neural network final task, we choose to work on a task based on Generative adversarial model(GAN) [2], namely Generative face completion [3]. Here, the GAN model is trained on real face images and the generator and discriminator network compete against one another. For this task, we decided to use CelebA dataset. The raw dataset is processed using OpenFace and divided into training, validation and test set. Once the model is trained on training set data, the test set images are masked to cover either random part of face or center part. The trained generative model is then used to fill the masked image with best suited facial structure. The project is an implementation of the idea presented in [3] and also uses few aspects of Semantic image inpainting [4].

In the next section, we will see different approach which has been identified to complete such complex task and their usefulness.

2. Related work and similar approach

Most of the classical image in-painting methods are based on information contained in the images like local or non-local information to fill the mask or complete the image. Examples of such approaches are Total variation (TV) [5], Low rank (LR) and PatchMatch (PM) [6] etc. All the previously mentioned approach assume that information like patches, pixels and structure; required to complete an image is present in the input image. But this is not the case when the patch to fill is complex and of arbitrary shape. As a consequence, these approaches proved to be efficient for filling small holes and patches but inefficient to recover the missing information for large patches. We can see few samples below of the previously mentioned approaches and their results [4]:



[Fig:1 Inpainting result from TV, LR and PM]

In order to fill large patches, it is evident that a trained neural network needs to consider the surrounding pixels or context. One of such approach named Context Encoder (CE)[1] became quite popular choice in past for such tasks. It suggests to use a neural network and trained it to encode the context information and then predict the masked patch. But CE results in unrealistic images as it considers the structural information during training but not during inference [4].

In recent years, approach based on GAN [2], Autoencoders (AEs) and Variational Autoencoders (VAEs) [7] gained popularity in similar tasks like interactive image editing and attribute based image editing tasks respectively. But GANs cannot be directly applied to image inpainting because they produce an entirely unrelated image with high probability, unless constrained by the provided masked image [4]. AEs and VAEs provide good results to learn complex distribution in unsupervised manner. Compared to GANs, VAEs tend to generate overly smooth images, which is not preferred for inpainting tasks. We can see few examples below for reference [4]:

[Fig:2 Images generated by a VAE and a DCGAN. First row: samples from a VAE. Second row: samples from a DCGAN]

In the next section, we will see the details of Semantic image inpainting approach [4] which has been used to complete the task of face completion and also its implementation details.

3. Theoretical basis and used procedures

We have done the face completion task based on Semantic image inpainting [4] approach which considers semantic inpainting as a constrained image generation problem and also incorporates the advantage of recent development in advances of GAN.

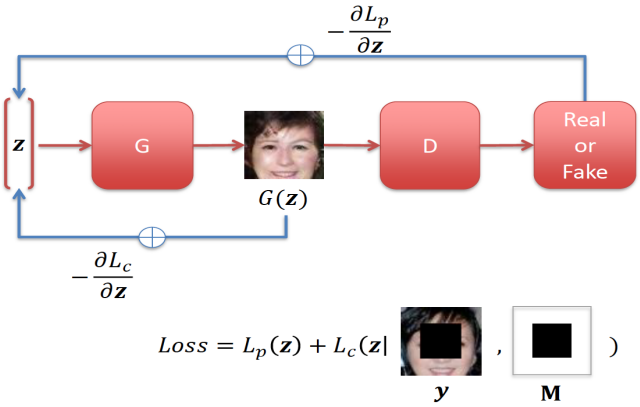
In this method, a deep generative model is trained which searches for an encoding of the masked image best suited to the image in latent space. Using this encoding, the generator is then used to reconstruct the masked images. In order to compare the best suited fill for the masked image, context-loss for masked image and prior-loss to penalize unrealistic images is used. Compared to the CE, one of the major advantages of this approach is that it does not require the masks for training and can be applied for arbitrarily structured missing regions during inference [4].

In order to fill the masked image with realistic image, both generator $\boldsymbol{G}$ and discriminator $\boldsymbol{D}$ is utilized and trained on the real image without the mask. Consider the original images are drawn from a distribution $\boldsymbol{p_{data}}$ and random sample $\boldsymbol{z}$ which will be used as input to G is drawn from the distribution $\boldsymbol{p_z}$. After training the G, it is expected that it will produce or mimic the samples from p_{data}. Once training is completed, it is expected that G will also be able to generate suitable recovered image from the masked sample.

3.1 Process flow and Loss formulation

There are two different types of loss functions namely Context loss(L_c) and Prior loss(L_p) are used in order to produce a suitable result for image recovery from the masked image. L_c and L_p are then back-propagated iteratively to produce the final result.

In order to understand the image recovery process better, it will be useful see the flow diagram below [4]:



[Fig.3: GAN model trained on real images, update z iteratively to find the closest mapping on the latent image manifold, based on the designed loss functions.]

We find the closest encoding for masked image as per the formulation presented in [4]. Please see the details below:

$$\hat{\mathbf{z}} = \arg\min_{\mathbf{z}} \{\mathcal{L}_c(\mathbf{z}|\mathbf{y}, \mathbf{M}) + \mathcal{L}_p(\mathbf{z})\},$$

where,

z-hat represents closest encoding,
y is the corrupted image,
M is the mask to represent the missing part of real image,
z is learned encoding manifold,
L_c is the contextual loss constrained on mask M and corrupted image y,
L_p is the prior loss which penalizes unrealistic images.

Basically, L_c gives emphasis on pixel level details and penalizes the model accordingly. L_1 norm is used for L_c which is calculated using the generated sample *G(z)* and an uncorrupted portion of the input image y. But instead of naive calculation, weight is given to the pixel depending on its distance from the mask. A pixel that is very far away from any holes plays a very little role in the image recovery process [4].

$$\mathcal{L}_c(\mathbf{z}|\mathbf{y}, \mathbf{M}) = \|\mathbf{W} \odot (G(\mathbf{z}) - \mathbf{y})\|_1.$$

L_p is identical to GAN loss for training the discriminator *D*. It mainly penalizes the model in order to produce images similar to images drawn from real images. Unlike L_c, L_p focuses on high-level image feature representation instead of pixel-wise difference [4].

$$\mathcal{L}_p(\mathbf{z}) = \lambda \log(1 - D(G(\mathbf{z}))).$$

3.2 Implementation details

Now, after defining the loss functions its time to implement our network and train it so that it could generate best-suited realistic image using randomly initialized z. With time, z is updated iteratively using back-propagation to minimize the overall loss.

The implementation uses architecture proposed in [8]. *G* takes a random 100-dimensional vector drawn from a uniform distribution between [-1, 1] and generates a 64x64x3 image. Transposed convolution layers are used to subsequently increase the dimension of the image. At each layer, the image size of doubled until it generates images of size 64x64x3. The discriminator model, *D*, is structured in reverse order. The input layer is an image of dimension 64x64x3, followed by a series of convolution layers where the number of channels is double the size of the previous layer, and the output layer is a two class soft-max.

For our task, we have only focused on center masking of real images and input image size after preprocessing has also been reduced to 64x64x3 due to the lack of computation power and limited scope of the project.

4. Network structure and design choices

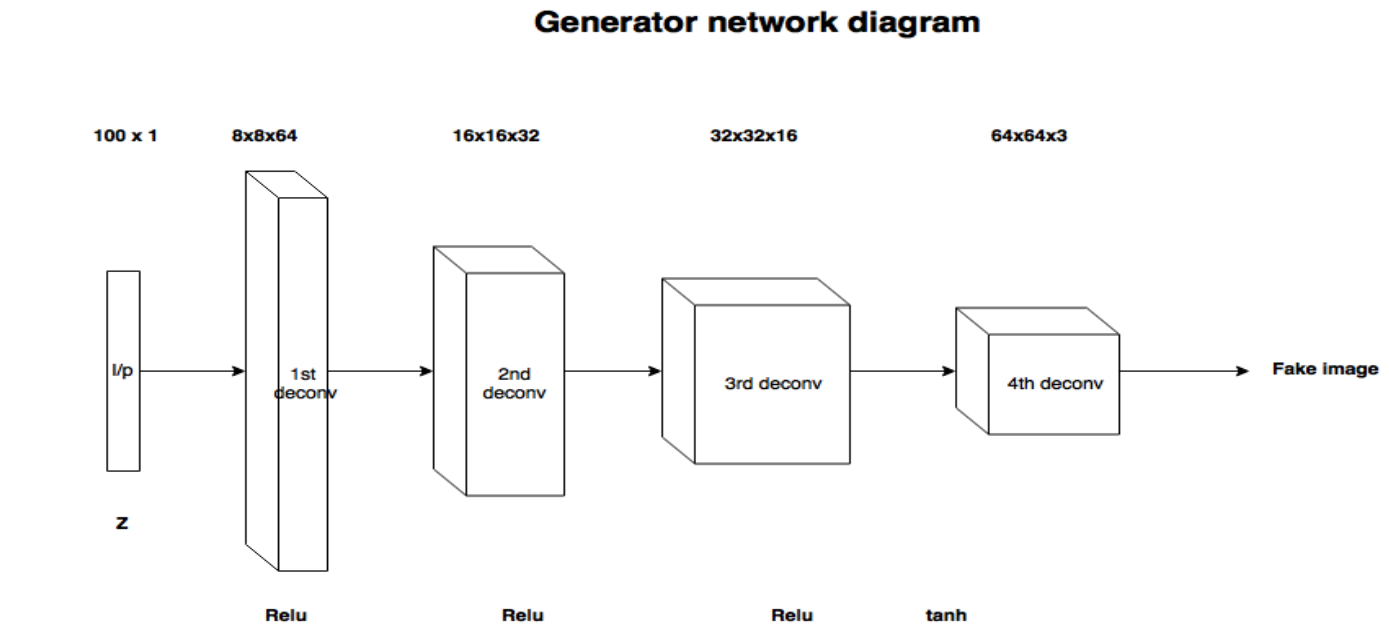
The computational network mainly consists of two components namely Generator ***G*** and Discriminator ***D***. *G* is designed to have single fully connected layer followed by four transposed convolution layers. For each transposed convolution layers we have used 5x5 filter size with default stride size of [2, 2]. The input to *G* is a sample drawn from a uniform distribution between [-1, 1]. At each layer, the image size is doubled to its input size and finally produce an image of size 64x64x3. The output of *G* is fed to the *D* along with the real images.

D network consists of four convolution layers of filter size 5x5 and default stride size [2, 2] and a single fully connected layer. We initialized the first convolution layer with 16 filters and doubled its quantity at every subsequent layers. The final convolution layer has 128 filters. Then, the output of convolution layers is flattened and connected to 64 fully connected layer. At last, the output of the discriminator is fed to a sigmoid function.

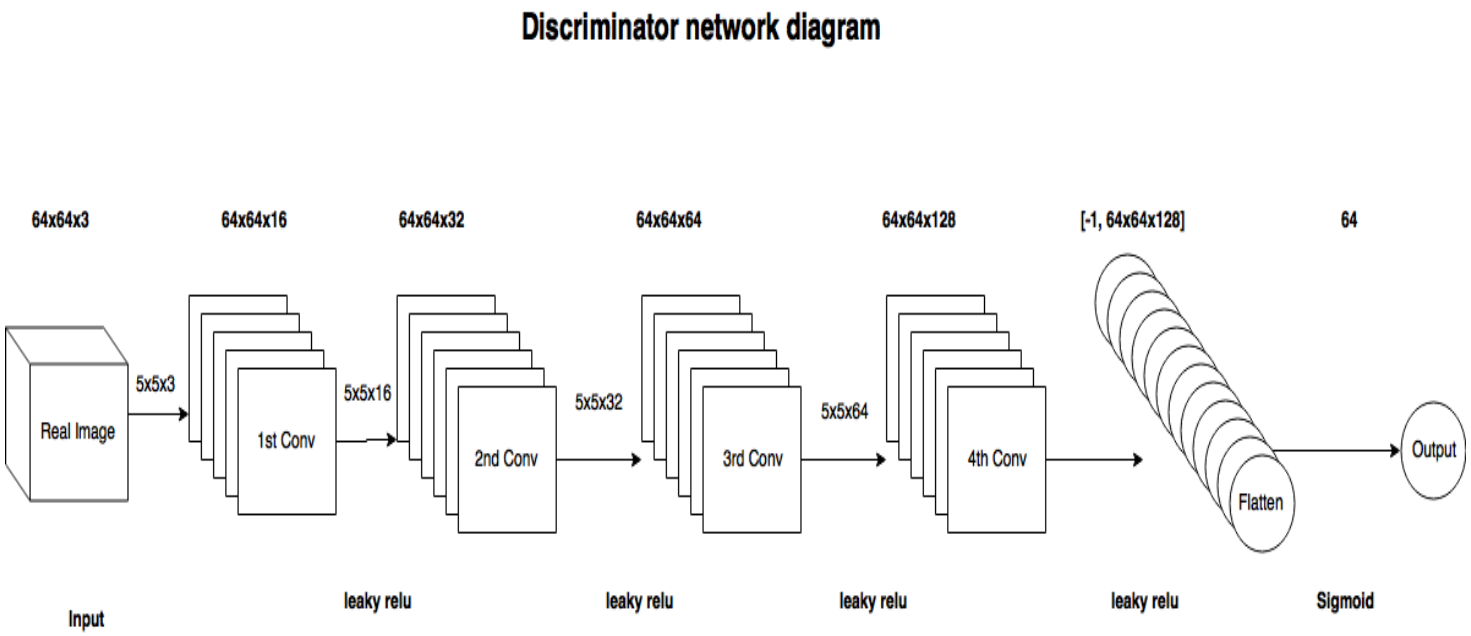
Batch normalization is used for both *D* and *G* during the training phase. *Leaky-relu* is used as activation function for *D* for all convolution layer and *relu* for *G* for every transposed convolution layer except for the final layer where *tanh* is used.

Choice of loss function has been described in detail in section 3.1. In order to optimize the loss function through back-propagation, we have used Adam gradient descent algorithm.

We tried complex network for G and D with varying activation functions on 5000 image samples. But, due to the complexity of the network and limited computational power, we couldn’t run it on entire training set. Out of all network we explored, the network structure drawn on the next page gave us comparable result on entire training set and we were also able to generate realistic images from the masked sample to a great extent.



[Fig:4 Network structure of Generator]

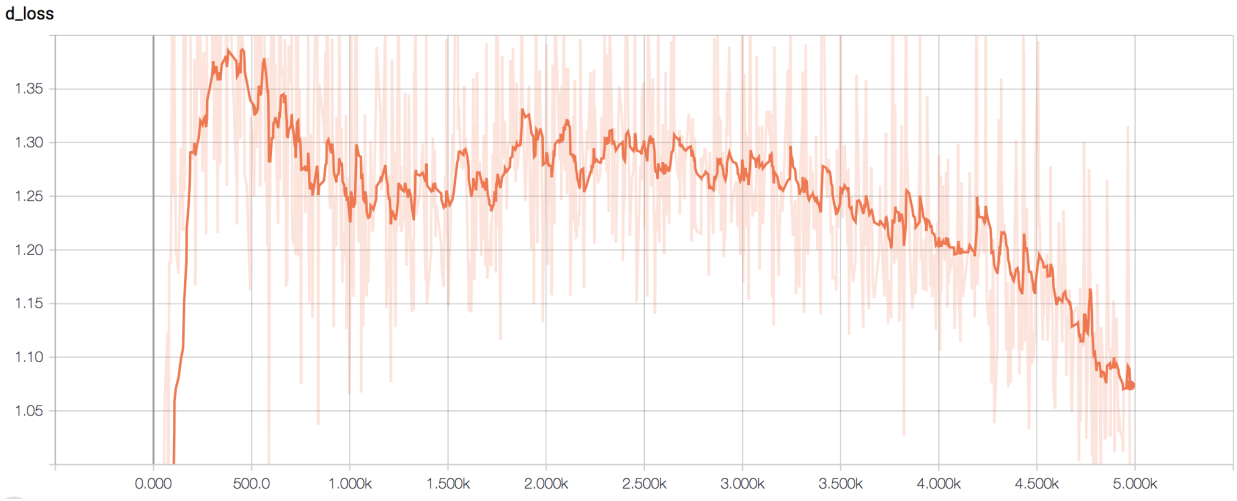


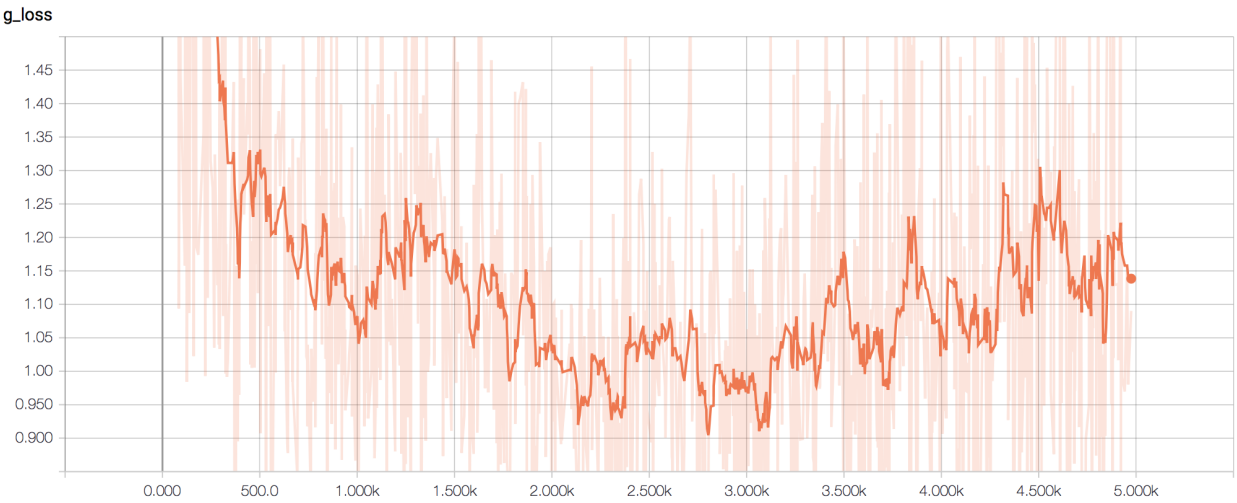
[Fig:5 Network structure of Discriminator]

5. Performance evaluation comparison and plots

We have evaluated the model performance by running the model on entire training data with batch size 300 and for 25 epochs. It was quite difficult for us to run the model by varying the batch size, learning rate and epochs on our system due to memory constraints. We tried to optimize the model performance by using validation data. After different run, we finally decided to train our model with 300 batch size, 0.0003 as learning rate, and 25 epochs.

Please see the plots for discriminator and generator for reference:

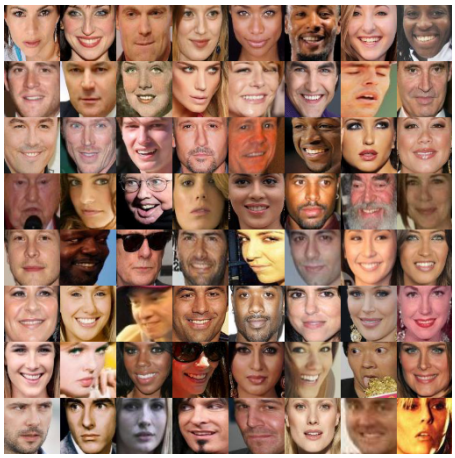




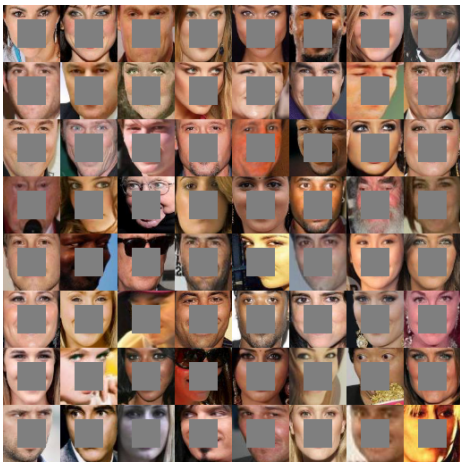
Also see the sample image generated by *G* at different iteration (0 to 4950):




It would make more sense if we could see and compare the original image, masked images and the recovered image generated by the model:



Original Image



Masked Image

Recovered image (450 iterations)

We can see that, the recovered images seem realistic but it can be improved with more complex models and also by using Poisson's blending as well to enhance the result.

6. Structure and layout

The implementation has entirely been done in *Python* (version 3.6) and *Tensorflow* (version 1.5). Apart from that, we have also used some other libraries like *Numpy*, *maths* etc. There are four main files that have been created for this implementation. All these files are present in *code* directory. The ***network.py*** file contains the implementation of Face_Completion model and also implementation for *D* and *G*. The ***helpers.py*** file contains all the helper functions which assist implementation of face completion task. Apart from these two files, we have other two python files namely ***face_completion.py*** and ***train_network.py*** which basically has argument parser used to call the Face_Completion class under network.py module by providing various parameters suitable as per requirement. We have taken reference of [9] for implementing complete function in the face_completion.py module.

We have also provided the tensorflow logs and checkpoints under *logs* and *checkpoint* directory. The *completed* directory contains the image samples which has been generated by face completion model on the masked images. In order to maintain brevity, we have only implement center masking for this task. The output samples are saved at every 50 intervals which can also be controlled in *face_completion.py* file through changing the argument "out_interval".

The processed-data.zip file contains the processed data used for training, validation and testing the model performance. We have used **CelebA** data which contain more than 200,000 images. Out of which we have only used 66,000 images divided in training (60,000), validation (5,000) and test (1,000) set. The raw data can be downloaded from link http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html. Once the raw data is downloaded, we can use OpenFace to process or align the data and also resize it to 64x64. (OpenFace is only compatible with Python 2.7). Use the below command to process the raw data (provided you have python 2.7 virtual environment, OpenFace and its dependencies installed on your system).

Code running instructions:

1. Data preprocessing (skip this step if preprocessed data is downloaded from myshare drive):

./openface/util/align-dlib.py /path/to/raw/data/ align innerEyesAndBottomLip /path/to/processdata/ --size 64

2. Training the network, epoch 25 and batch size 300.

python train_network.py --dataset /processed-data/train/ --epoch 25 --batch_size 300

3. Running the face completion module to fill the masked image.

python face_completion.py /processed-data/test/ --out_dir completed*

The images from the test set are masked at center on the fly and these masked images are filled with appropriate facial structure and stored in the completed directory. The samples are generated at an interval of 50 so that we could see the progress of our model regularly.

We don't expect any problem running and checking our model performance but in case of any issues, please contact git repository owner (Manish Kumar).

References

[1] D. Pathak, P. Krahenbuhl, J. Donahue, T. Darrell, and A. A. Efros, "Context Encoders: Feature Learning by Inpainting," 2016.

[2] I. Goodfellow *et al.*, "Generative Adversarial Nets," *Adv. Neural Inf. Process. Syst. 27*, pp. 2672–2680, 2014.

[3] Y. Li, S. Liu, J. Yang, and M.-H. Yang, "Generative Face Completion," 2017.

[4] R. A. Yeh, C. Chen, T. Y. Lim, A. G. Schwing, M. Hasegawa-Johnson, and M. N. Do, "Semantic Image Inpainting with Deep Generative Models," 2016.

[5] J. Shen and T. F. Chan, "Mathematical Models for Local Nontexture Inpaintings," *SIAM J. Appl. Math.*, vol. 62, no. 3, pp. 1019–1043, 2002.

[6] C. Barnes, D. B. Goldman, E. Shechtman, and A. Finkelstein, "The PatchMatch randomized matching algorithm for image manipulation," *Commun. ACM*, vol. 54, no. 11, p. 103, 2011.

[7] D. P. Kingma and M. Welling, "Auto-Encoding Variational Bayes," no. Ml, pp. 1–14, 2013.

[8] A. Radford, L. Metz, and S. Chintala, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks," pp. 1–16, 2015.

[9] http://bamos.github.io/2016/08/09/deep-completion/