IMAGE MODELING USING GENERATIVE ADVERSARIAL NETWORK

Project report submitted in partial fulfilment of the requirement for the degree of Bachelor of Technology

In

Computer Science Engineering

Submitted By:

Vishrut Thakur (171309)

under the supervision of

Dr. Rajinder Sandhu

To

Department of Computer Science & Engineering and Information Technology,

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY,

Waknaghat, Himachal Pradesh-173234

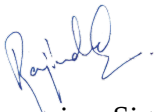
Candidate's Declaration

I hereby declare that the work presented in this report entitled **"IMAGE MODELING USING GENERATIVE ADVERSARIAL NETWORK"** in partial fulfilment of the requirements for the award of the degree of **Bachelor of Technology** in **Computer Science and Engineering** submitted in the department of Computer Science & Engineering and Information Technology, Jaypee University of Information Technology Waknaghat is an authentic record of my own work carried out over a period from August 2020 to December 2020 under the supervision of **Dr. Rajinder Sandhu**, Assistant Professor (Senior Grade), Computer Science and Engineering/Information Technology).

The matter embodied in the report has not been submitted for the award of any other degree or diploma.

(Student Signature)
Vishrut Thakur, 171309

This is to certify that the above statement made by the candidate is true to the best of my knowledge.



(Supervisor Signature)
Dr Rajinder Sandhu
Assistant Professor (Senior Grade)
Computer Science and Engineering/Information Technology
Dated:

Acknowledgement

Any serious and lasting achievement cannot be achieved without the help, guidance and co-operation of numerous people involved in the work. I thank the almighty for giving me the courage and perseverance in completing this-project.

I extend my sincere thanks to Prof. Dr. VINOD KUMAR Vice Chancellor of our University, for providing sufficient infrastructure and good environment in the University to complete our course.

I am thankful to our Registrar and Dean of Students Maj Gen RAKESH BASSI (Retd.), for providing the necessary Infrastructure and labs and also permitting to carry out this project.

With extreme jubilance and deepest gratitude, I would like to thank Director & Head of the C.S.E. Department, Prof. Dr. SAMIR DEV GUPTA for his constant encouragement.

I special thanks to our Project coordinator DR. HEMRAJ SAINI, Associate Professor Computer Science & Engineering, for his support and valuable suggestions regarding project work.

I am greatly indebted to project guide DR. RAJINDER SANDHU, Assistant Professor (Senior Grade), Computer Science & Engineering Department, for providing valuable guidance at every stage of this project work. I am profoundly grateful towards the unmatched services rendered by him.

My special thanks to all the faculty of Computer Science & Engineering and peers for their valuable advises at every stage of this work.

Last but not least, I would like to express my deep sense of gratitude and earnest thanks giving to my dear parents for their constant moral support and heartfelt cooperation at every step of life.

Table of Contents

List of Abbreviations

- CNN - Convolution Neural Networks.
- DCGANs - deep convolutional generative adversarial networks.
- MNIST - Modified National Institute of Standards and Technology.
- SVHN – Street View House Numbers
- CIFAR- Canadian Institute for Advanced Research
- VGG – Visual Geometry Group
- FPR – False Positive Rate
- LSUN - Large scale images showing different objects from given categories like bedroom, tower, etc
- SVBRDF - Spatially-Varying Bi-Directional Reflectance Distribution Functions.

List of Figures

List of Tables

Abstract

Amongst recent research work in Computer Vision Applications, supervised learning with CNNs have been highly popular. Unfortunately, this can't be inferred for its counterpart in unsupervised learning.

Here, in this project I introduce another anecdote of CNNs called Deep Convolutional Generative Adversarial Networks more commonly abbreviated as DCGANs. DCGANs having distinctive characteristics in-order can proof its potential among other classes of unsupervised learning. Training on various image datasets, we show convincing evidence that our deep convolutional adversarial pair learns a hierarchy of representations from object parts to scenes in both the generator and discriminator. Additionally, we use the learned features for novel tasks - demonstrating their applicability as general image representations.

CHAPTER 1 - INTRODUCTION

1.1 Introduction

Various supervised learning tasks like image classification can be performed by consummating the vast variety of unlabelled representations and therefore generating good intermediate representations. As we know in todays research work learning reusable representation from large unlabelled datasets has been very popular.

"We propose that one way to build good image representations is by training Generative Adversarial Networks (GANs)[1]", and later reusing parts of the generator and discriminator networks as feature extractors for supervised tasks. GANs provide an attractive alternative to maximum likelihood techniques. One can additionally argue that their learning process and the lack of a heuristic cost function (such as pixel-wise independent mean-square error) are attractive to representation learning. GANs have been known to be unstable to train, often resulting in generators that produce nonsensical outputs.

1.2 Objective

To understand and visualize what GANs learn, and the intermediate representations of multi-layer GANs. We capitalize on large amounts of unlabeled images in order to learn a model of scene dynamics for image generation task. We propose a generative adversarial network for image with a convolutional architecture that untangles the image foreground from the background. Experiments suggest this model can generate image better than simple baselines, and we show its utility at predicting futures of static images.

Moreover, experiments and visualizations show the model internally learns useful features for recognizing actions with minimal supervision, suggesting scene dynamics are a promising signal for representation learning. We believe generative image models can impact many applications in image understanding and simulation.

1.3 Methodology

- We suggest and compare a fixed of constraints at the architectural topology of Convolutional GANs that make them stable to educate in most settings. We name this

class of architectures “Deep Convolutional Generative Adversarial Networks (DCGANs)”

- We use the trained generators for image classification tasks, showing competitive performance with other unsupervised algorithms.
- We visualize the filters learnt by GANs and empirically show that specific filters have learned to draw specific objects.
- We show that the generators have interesting vector arithmetic properties allowing for easy manipulation of many semantic qualities of generated samples.
- We use the trained discriminators for image classification tasks, showing competitive performance with other unsupervised algorithms.

CHAPTER 2 - LITERATURE SURVEY

2.1 Related Work

2.1.1 Representation Learning from Unlabelled Data

Talking about unsupervised representation learning, It has been a fairly well studied problem in general computer vision research, as well as in the context of images. A classic approach to unsupervised representation learning is to do clustering on the data (for example using K-means), and leverage the clusters for improved classification scores. “In the context of images, one can do hierarchical clustering of image patches[2] to learn powerful image representations. Another popular method is to train auto-encoders (convolutionally, stacked[3], separating the what and where components of the code[4], ladder structures[5])” that encode an image into a compact code, and decode the code to reconstruct the image as accurately as possible. These methods have also been shown to learn good feature representations from image pixels. “Deep belief networks[6] have also been shown to work well in learning hierarchical representations.”

2.1.2 Generating Natural Images

“Generative image models are well studied and fall into two categories:

- parametric
- non-parametric.”

“The non-parametric models often do matching from a database of existing images, often matching patches of images, and have been used in texture synthesis[7] super-resolution[8] and in- painting[9].”

There has been drastic exploration of parametric models for generating images (“for example on MNIST digits or for texture synthesis[10]”. However, generating natural images of the real world have had not much success until recently. “A variational sampling approach to generating images[11] has had some success, but the samples often suffer from being blurry. Another approach generates images using an iterative forward diffusion process[12].”

“Generative Adversarial Networks[13] generated images suffering from being

noisy and incomprehensible". "A Laplacian pyramid extension to this approach[14] showed higher quality images, but they still suffered from the objects looking wobbly because of noise introduced in chaining multiple models." "A recurrent network approach[15] and a deconvolution network approach[16] have also recently had some success with generating natural images." But generators were never used for supervised tasks.

2.1.3 Visualizing the Internals of CNNs

One constant criticism of using neural networks has been that they are black-box methods, with little understanding of what the networks do in the form of a simple human-consumable algorithm. In the context of CNNs[17], (Zeiler & Fergus, 2014) showed that by using deconvolutions and filtering the maximal activations, one can find the approximate purpose of each convolution filter in the network. "Similarly, using a gradient descent on the inputs lets us inspect the ideal image that activates certain subsets of filtersmord[18]."

2.2 Related Work

2.2.1 Parametric Texture Synthesis

Some early methods for texture synthesis explored parametric models. "[19] used histogram matching combined with Laplacian and steerable pyramids to synthesize textures." We are inspired by their use of histogram matching. "[10]investigated the integration of many wavelet statistics over different locations, orientations, and scales into a sophisticated parametric texture synthesis method." These included cross-correlations between pairs of filter responses.

2.2.2 Neural Texture Synthesis and Style Transfer

In this paper, for short, we use *neural* to refer to convolutional neural networks. Recently, "[20] showed that texture synthesis can be performed by using ImageNet-pretrained convolutional neural networks such as VGG[21]." Specifically, "[20]impose losses on co-occurrence statistics for pairs of features." These statistics are computed via Gram matrices, which measure inner products between all pairs of feature maps within the same layers of the CNN.

Results of[20] were typically better than those of[10] for texture synthesis. Later extended

this approach to style transfer, by incorporating within CNN layers both a *Forbinius norm content loss* to a content exemplar image, and a *Gram matrix style loss* to a style exemplar image. We build upon this framework, and offer a brief review of how it works in the next section. Concurrently to our research,(Berger et al, 2016). observed that in the approach of[20], texture regularity may be lost during synthesis, and proposed a loss that improves regularity based on co-occurrence statistics between translations of the feature maps.

"Recently, [22] used neural networks to extract SVBRDF material models from a single photo of a texture. Their method focuses on a more specific problem of recovering a SVBRDF model from a head-lit flash image." However, they do observe that positive instabilities along with non-stationary textures can effortlessly end result if sufficiently informative information are not used. We see this as related with our observations about instabilities and the way to repair them.

2.2.3 Feedforward Neural Texture Synthesis

"Recently, a few papers ([23][24]) have investigated the training of feedforward synthesis mod- els, which can be pre-trained on a given exemplar texture or style, and then used to quickly synthesize a result using fixed network weights." The feed-forward strategy is faster at run-time and uses less memory. However, feed-forward methods must be trained specifically on a given style or texture, making the approach impractical for applications where such a pre-training would take too long (pre-training times of 2 to 4 hours are reported in these papers).

2.2.4 Non-Parametric Texture Synthesis

"Non-parametric texture syn- thesis methods work by copying neighborhoods or patches from an exemplar texture to a synthesized image according to a local similarity term[7]"; Wei and Levoy 2000; Lefebvre and Hoppe 2005; Lefebvre and Hoppe 2006; Kwatra et al. 2003; Kwatra et al. 2005; Barnes et al. 2009]. This approach has also been used to transfer style [Efros and Freeman 2001; Hertzmann et al. 2001; Barnes et al. 2015]. Some papers have recently combined parametric neural network models with non- parametric patch-based models [Chen and Schmidt 2016; Li and Wand 2016].

CHAPTER 3 - SYSTEM DEVELOPMENT

3.1 Approach

- All convolutional net (Springenberg et al., 2014) which replaces deterministic spatial pooling functions (such as maxpooling) with strided convolutions, allowing the network to learn its own spatial downsampling. We use this approach in our generator, allowing it to learn its own spatial upsampling, and discriminator.
- Trend towards eliminating fully connected layers on top of convolutional features. The strongest example of this is global average pooling which has been utilized in state of art image classification models (Mordvintsev et al.). We found global average pooling increased model stability but hurt convergence speed. A middle ground of directly connecting the highest convolutional features to the input and output respectively of the generator and discriminator worked well. The first layer of the GAN, which takes a uniform noise distribution Z as input, could be called fully connected as it is just a matrix multiplication, but the result is reshaped into a 4-dimensional tensor and used as the start of the convolution stack. For the discriminator, the last convolution layer is flattened and then fed into a single sigmoid output.
- Batch Normalization which stabilizes learning by normalizing the input to each unit to have zero mean and unit variance. This helps deal with training problems that arise due to poor initialization and helps gradient flow in deeper models. This proved critical to get deep generators to begin learning, preventing the generator from collapsing all samples to a single point which is a common failure mode observed in GANs. Directly applying batchnorm to all layers, resulted in sample oscillation and model instability.

3.2 The ReLU Activation

It is used in the generator with the exception of the output layer which uses the Tanh function. We observed that using a bounded activation allowed the model to learn more quickly to saturate and cover the color space of the training distribution. Within the discriminator we found the leaky rectified activation to

work well.

Architecture guidelines for stable Deep Convolutional GANs

- Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).
- Use batchnorm in both the generator and the discriminator.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in generator for all layers except for the output, which uses Tanh.
- Use Leaky ReLU activation in the discriminator for all layers.

3.3 Details Of Adversarial Training

We trained DCGANs on three datasets, Large-scale Scene Understanding (LSUN) Imagenet-1k and a newly assembled Faces dataset. Details on the usage of each of these datasets are given below.No pre-processing was applied to training images besides scaling to the range of the tanh activation function [-1, 1]. All models were trained with mini-batch stochastic gradient descent (SGD) with a mini-batch size of 128.Weights were initialized from a zero-centered Normal distribution with standard deviation 0.02. In the LeakyReLU, the slope of the leak was set to 0.2 in all models. We found the suggested learning rate of 0.001, to be too high, using 0.0002 instead. We found leaving the momentum term $\beta 1$ at suggested value of 0.9 resulted in training oscillation and instability while reducing it to 0.5 helped stabilize training.

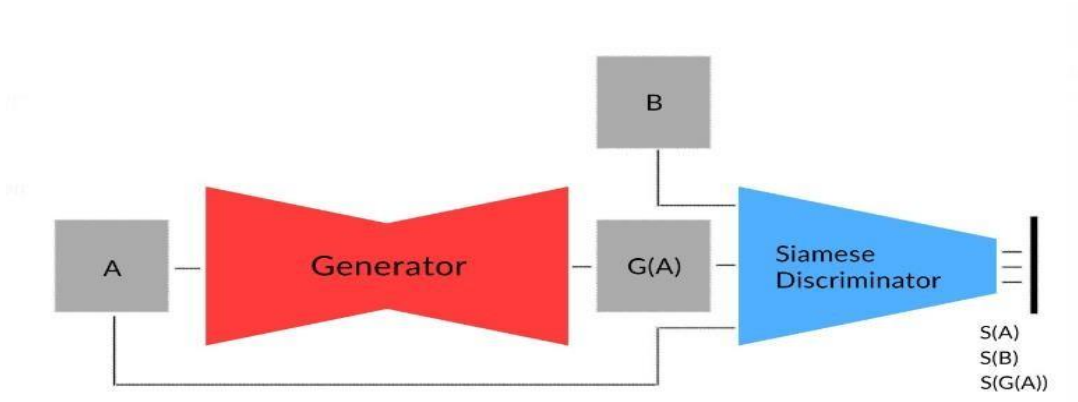


Figure 3.1: Siamese GAN architecture.

It is made of a single generator (G) and discriminator (D): G takes an image as input and

outputs the translated image; D takes an image as input and outputs a latent vector.

The Siamese Discriminator has 2 objectives: telling G how to generate more realistic images and maintaining in those fake images a correlation (same 'content') with the original ones.

Calling A1, A2 and B1, B2 random images from domains A and B respectively, X a random image, and G(X) the images generated by the Generator, the Discriminator must encode images into vectors D(X) such as:

- D(B1) must be close (euclidean distance) to a fixed point (the origin for example), while D(G(A1)) must be far from the same point. Consequently, vectors closer to the fixed points represent more realistic images. The Generator on the other hand tries to minimize the distance from D(G(A1)) to the fixed point, in classic adversarial fashion.
- (D(A1)-D(A2)) must be similar (cosine similarity) to (D(G(A1))-D(G(A2))), to preserve 'content' between A and G(A). Both the Generator and the Discriminator take part in this objective.

3.3.1 Deduplication

To similarly lower the likelihood of the generator memorizing input examples (Figure 3.2) A simple image de-duplication process is performed over the input sample. We fit a 3072-128-3072 de-noising dropout regularized RELU autoencoder on 32x32 down sampled center-crops of training examples. "The resulting code layer activations are then binarized via thresholding the ReLU activation which has been shown to be an effective information preserving technique[25]" and provides a convenient shape of semantic-hashing, bearing in mind linear time de-duplication. visible inspection of hash collisions showed excessive precision with an estimated FPR of much less than 1 in 100. Additionally, the technique detected and removed approximately 275,000 near duplicates, suggesting a high recall.

3.4 Faces

Dbpedia was using to obtain random people's names, further these names were used to scrap images having human face from randomized web queries with only conditioned for having born in recent times. This dataset has 3M images from 10K people. OpenCV face detector was used over this dataset, only those detected images were kept which had

adequate resolution, this came out to be 350,000 face boxes. Now, these face boxes will be used for training the sampler model, while keeping in mind that no data augmentation was processed on the images.

3.5 Imagenet-IK

"We use Imagenet-1k[26] as a source of natural images for unsupervised training." 32 × 32 min-resized center crops were used to train but again no data augmentation was performed.

Figure 3.2 shows the result after one training pass. Theoretically, the model could learn to memorize training examples, but this is experimentally unlikely as we train with a small learning rate and minibatch SGD. We are aware of no prior empirical evidence demonstrating memorization with SGD and a small learning rate.

Figure 3.2: Generated bedrooms after one training pass through the dataset.

Figure 3.3 shows resultant images after 5 successful epochs of training. Signs of visual under-fitting is quite evident due to consistent noise textures across multiple samples like the base boards of some of the beds.

Figure 3.3: Generated bedrooms after five epochs of training.

CHAPTER 4 - PERFORMANCE ANALYSIS

4.1 Investigation and Visualization of The Internals of The Networks

- We investigate the trained generators and discriminators in a variety of ways.
- We do not do any kind of nearest neighbor search on the training set. Nearest neighbors in pixel or feature space are trivially fooled (Theis et al., 2015) by small image transforms. We also do not use log-likelihood metrics to quantitatively assess the model, as it is a poor (Theis et al., 2015) metric.

Table 1: SVHN classification with 1000 labels

Model	Error Rate
KNN	77.93%
DCGAN (ours) + L2-SVM	22.48%
Supervised CNN with the same architecture	28.87% (validation)

4.1.1 The Latent Space

Prima facia investigation was done to figure out the basic structure of the latent space. Walking on the manifold that is learnt can generally informs us about the possible signs of memorization (like if we observe any sharp transitions) and hierarchical collapse of space. If taking walks on this latent area consequences in semantic modifications to the picture generations, a decent argument can be made on the model if relevant and worthwhile representations were made. The results are shown in Figure 4.1. The first few rows demonstrate that the learned space has smooth transitions. While in 6^{th} and 10^{th} rows the picture is totally opposite, it is visible that in 6^{th} row a room with no window and only a scar on wall slowly converts to a room with a large window and similarly in row 10 a room with television slowly gets a window in place of the TV.

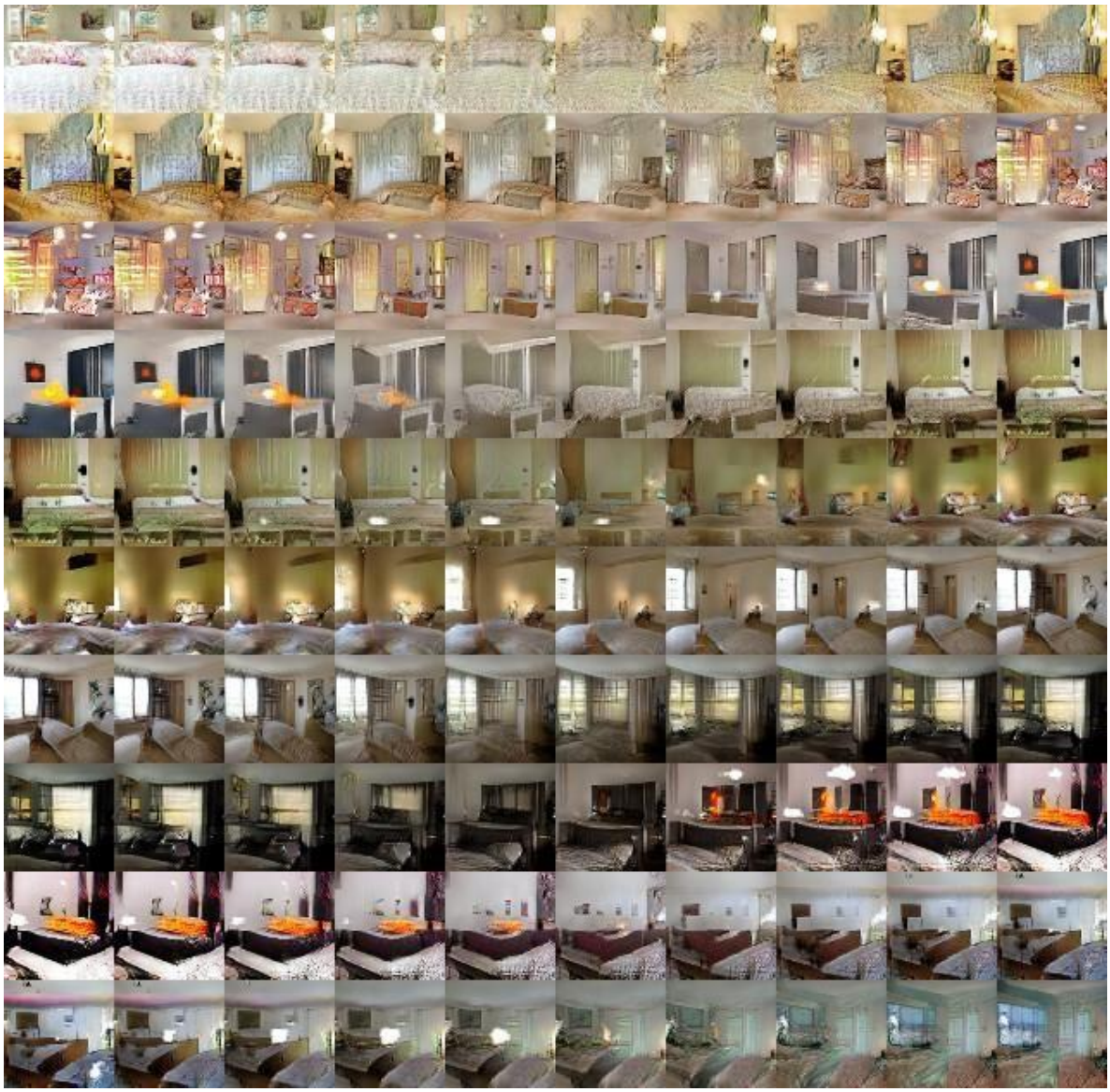


Figure 4.1: Top rows: Interpolation between a series of 9 random points in Z show that the space learned has smooth transitions, with every image in the space plausibly looking like a bedroom. In the 6th row, you see a room without a window slowly transforming into a room with a giant window. In the 10th row, you see what appears to be a TV slowly being transformed into a window.

4.2 Visualization of the Discriminator Features

"Previous work has demonstrated that supervised training of CNNs on large image datasets results in very powerful learned features[17]." In addition to it, a traditional CNN can be trained on classification to work as a object detector. Here I'm going to demonstrate how an unsupervised model like that of DCGAN if trained on a large enough image dataset can

also work on the generate a hierarchy of interesting features. “Using guided backpropagation as proposed by[27], that the features learnt by the discriminator activate on typical parts of a bedroom, like beds and windows.” Therefore, to draw a comparison between these approaches in (Figure 4.2), we supply a baseline for randomly initialized capabilities that are not activated on anything that is semantically applicable or interesting.

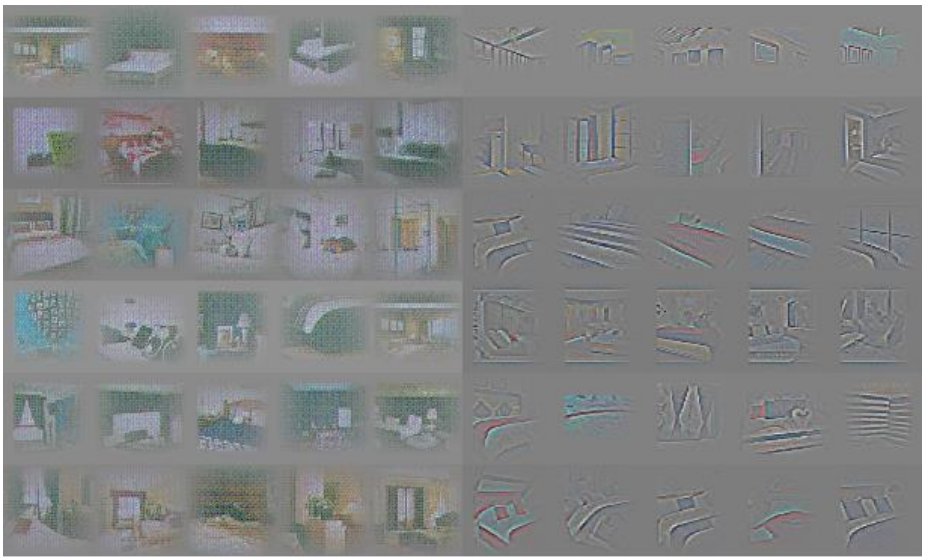


Figure 4.2: On the right, guided backpropagation visualizations of maximal axis-aligned responses for the first 6 learned convolutional features from the last convolution layer in the discriminator. Notice a significant minority of features respond to beds - the central object in the LSUN bedrooms dataset. On the left is a random filter baseline. Comparing to the previous responses there is little to no discrimination and random structure.

4.3 Manipulating the Generator Representation

4.3.1 Forgetting to Draw Certain Objects

In addition to the representations learnt by a discriminator, there is the question of what representations the generator learns. The quality of samples suggest that the generator learns specific object representations for major scene components such as beds, windowsand miscellaneous furniture. In order to explore the form that these representations take, we conducted an experiment to attempt to remove windows from the generator completely.

On 150 samples, 52 window bounding boxes were drawn manually. On the second highest convolution layer features, logistic regression was fit to predict whether a feature activation was on a window, by using the criterion that activations inside the

drawn bounding boxes are positives and random samples from the same images are negatives. Using this simple model, all feature maps with weights greater than zero were dropped from all spatial locations. Then random new samples were generated with and without the feature map removal. The generated images with and without the window dropout are shown in Figure 4.3, and interestingly, the network mostly forgets to draw windows in the bedrooms, replacing them with other objects.

Figure 4.3: Top row contains un-modified samples while Bottom row shows generated outputs after dropping out *window filters.*

4.3.2 Vector arithmetic on face samples

In the context of evaluating learned representations of words (Mikolov et al., 2013) demonstrated that simple arithmetic operations revealed rich linear structure in representation space. One canoni- cal example demonstrated that the vector(”King”) - vector(”Man”) + vector(”Woman”) resulted in a vector whose nearest neighbor was the vector for Queen. We investigated whether similar structure emerges in the *Z* representation of our generators. We performed similar arithmetic on the *Z* vectors of sets of exemplar samples for visual concepts.

Experiments working on only single samples per concept were unstable, but averaging the *Z* vector for three examplars showed consistent and stable generations that semantically obeyed the arithmetic. In addition to the object manipulation shown in Figure 4.4 , we demonstrate that face pose is also modeled linearly in *Z* space Figure 4.6.

These demonstrations suggest interesting applications can be developed using *Z* representations learned by our models. It has been previously demonstrated that conditional generative models can learn to convincingly model object attributes like scale, rotation, and position[16]. This is to our knowledge the first demonstration of this occurring in purely unsupervised models.

In addition, exploring and developing the above-mentioned vector arithmetic should dramatically reduce the amount of information wanted for conditional generative modeling of complicated photo distributions.

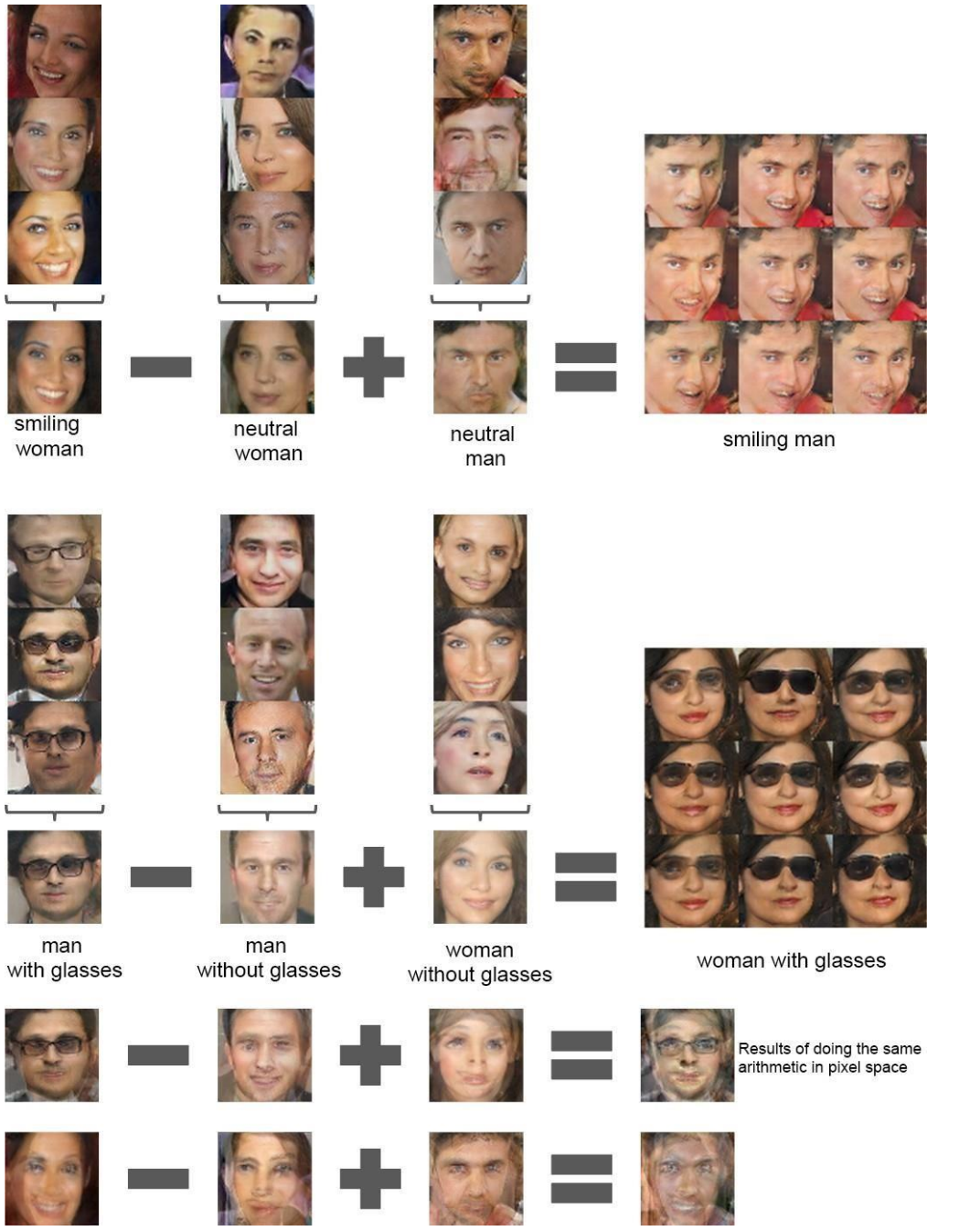


Figure 4.4: Vector arithmetic for visual concepts. For each column, the Z vectors of samples are averaged. Arithmetic was then performed on the mean vectors creating a new vector Y . The center sample on the right-hand side is produce by feeding *Y* as input to the generator. To demonstrate the interpolation capabilities of the generator, uniform

noise sampled with scale +-0.25 was added to *Y* to produce the 8 other samples. Applying arithmetic in the input space (bottom two examples) results in noisy overlap due to misalignment.

Input

Figure 4.5: Input to Model

Output

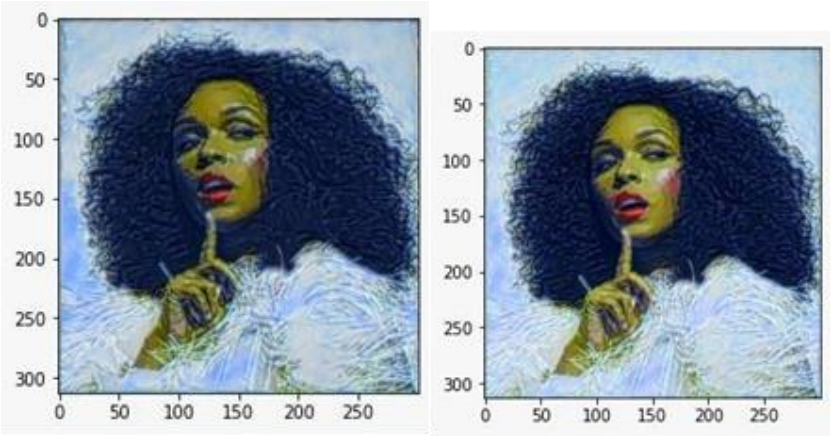


Figure 4.6: Output Generated From Model

4.4 Classifying CIFAR-10 Using GANs as a Feature Extractor

"One common technique for evaluating the quality of unsupervised representation learning algorithms is to apply them as a feature extractor on supervised datasets and evaluate the performance of linear models fitted on top of these features.[28]"

On the CIFAR-10 dataset, a very strong baseline performance has been demonstrated from a well-tuned single layer feature extraction pipeline utilizing K-means as a feature learning algorithm. When using a very large amount of feature maps (4800) this ×itechnique

achieves 80.6% accuracy. “An unsupervised multi-layered extension of the base algorithm reaches 82.0% accuracy[29].” to evaluate the satisfactory of the representations found out by× DCGANs for supervised duties, we teach on Imagenet-1k and then use the discriminator’s convolutional capabilities from all layers, maxpooling every layer’s representation to produce a 4*4 spatial grid. These features are then flattened and concatenated to form a 28672-dimensional vector and a regularized linear L2-SVM classifier is trained on top of them. This achieves 82.8% accuracy, outperforming all K-means based approaches. Notably, the discriminator has many less feature maps (512 in the highest layer) compared to K-means based techniques, but does result in a larger total feature vector size due to the many layers of 4*4 spatial locations. “The performance of DCGANs is still less than that of Exemplar CNNs[16]” , a technique which trains normal discriminative CNNs in an unsupervised style to distinguish among specifically chosen, aggressively augmented, exemplar samples from the source dataset. similarly, might be made with the aid of finetuning the discriminator’s representations, however we depart this for future work. additionally, on the grounds that our DCGAN changed into never skilled on CIFAR-10 this experiment also demonstrates the area robustness of the learned capabilities.

Table 2: CIFAR-10 classification results using our pre-trained model. Our DCGAN is not pre- trained on CIFAR-10, but on Imagenet-1k, and the features are used to classify CIFAR-10 images.

Model	**Accuracy**	**Accuracy (400per class)**	**Max #of features units**
1 Layer K-means	80.6%	63.7% (±0.7% Error)	4800
3 Layer K-means Learned	82.0%	70.7% (±0.7%Error)	3200
RF		72.6% (±0.7%Error)	
View Invariant K-means	81.9%	77.4% (±0.2%Error)	6400
Exemplar CNN	84.3%		1024
DCGAN (ours) + L2-SVM	82.8%	73.8% (±0.4%Error)	512

4.5 Classifying SVHN Digits Using GANs As A Feature Extractor

"On the StreetView House Numbers dataset (SVHN)[1], we use the features of the discriminator of a DCGAN for supervised purposes when labeled data is scarce." Following comparable dataset practice rules as in the CIFAR-10 experiments, we break up off a validation set of 10,000 examples from the non-more set and use it for all hyperparameter and version choice. 1000 uniformly class allotted training examples are randomly decided on and used to educate a regularized linear L2-SVM classifier on top of the same feature extraction pipeline used for CIFAR-10. "Additionally, we validate that the CNN architecture used in DCGAN is not the key contributing factor of the model's performance by training a purely supervised CNN with the same architecture on the same data and optimizing this model via random search over 64 hyperparameter trials [30]." It achieves a significantly higher 28.87% validation error.

4.6 Evaluating DCGANs Capability To Capture Data Distributions

We propose to apply standard classification metrics to a conditional version of our model, evaluating the conditional distributions learned. We trained a DCGAN on MNIST (splitting off a 10K validation set) as well as a permutation invariant GAN baseline and evaluated the models using a nearest neighbor classifier comparing real data to a set of generated conditional samples. We found that removing the scale and bias parameters from batch norm produced better results for both models. We speculate that the noise introduced by batch norm helps the generative models to better explore and generate from the underlying data distribution. The results are shown in Table 3 which compares our models with other techniques. The DCGAN model achieves the same test error as a nearest neighbor classifier fitted on the training dataset - suggesting the DCGAN model has done a superb job at modelling the conditional distributions of this dataset. "At one million samples per class, the DCGAN model outperforms InfiMNIST[5]", a hand developed data augmentation pipeline which uses translations and elastic deformations of training examples. "The DCGAN is competitive with a probabilistic generative data augmentation technique utilizing learned per class transformations[14] while being more general as it directly models the data instead of transformations of the data."

Table 3: Nearest neighbor classification results.

Model	Test Error @50K samples	Test Error @10M samples
AlignMNIST		1.4%
InfiMNIST		2.6%
Real Data	3.1%	
GAN	6.28%	5.65%
DCGAN (ours)	2.98%	1.48%

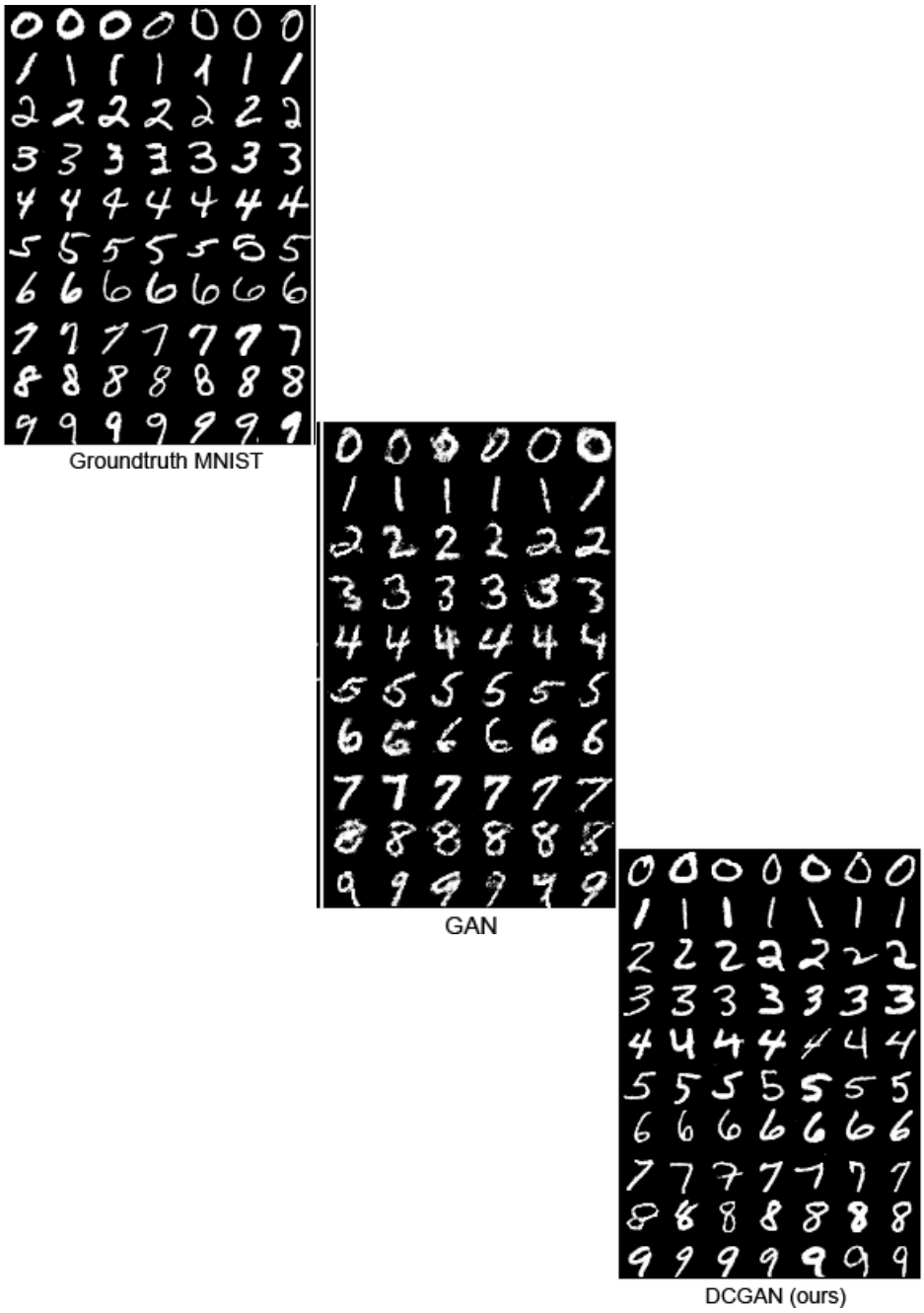


Figure 4.7: Side-by-side illustration of (from left-to-right) the MNIST dataset, generations from a baseline GAN, and generations from our DCGAN

Figure 4.8: More face generations from our Face DCGAN.

Figure 4.9: Generations of a DCGAN that was trained on the Imagenet-1k dataset.

We now discuss some blessings of our end result. Our histogram loss addresses instabilities through ensuring that the total statistical distribution of the capabilities is preserved. further to enhancing photo pleasant, our complete loss (inclusive of the

histogram loss) also calls for fewer iterations to converge. “We use a mean of 700 iterations for our results, which we find consistently give good quality, and 1000 iterations for the results of [24]”, This we find can be unstable at time. “We note that methods based on Gram matrices such as [20] can become unstable over the iteration count.” We interpret this as being caused by the mean and variance being free to drift, as we discussed in 4.2.

By adding histograms to our loss function, the result is more stable and converges better, both spatially and over iterations.

Running times for our method are as follows. We used a machine with four physical cores (Intel Core i5-6600k), with 3.5 GHz, 64 GB of RAM, and an Nvidia Geforce GTX1070 GPU with 8 GB of GPU RAM, running ArchLinux. For a single iteration on the CPU, our method takes 7 minutes and 8 seconds, whereas [20] takes 15 minutes 35 seconds. This equates to our method requiring only 45.7% of the running time for the original Gatys method. Our approach used three pyramid levels and histogram loss at relu4 1 and relu1 1. These metrics were measured over 50 iterations synthesizing a 512x512 output. We currently have most but not all of our algorithm implemented on the GPU. Because not all of it is implemented on the GPU, a speed comparison with our all-GPU implementation of[20] is not meaningful, therefore we ran both approaches using CPU only.

Here we present two simple methods to preserve the color of the content image in the neural style transfer algorithm:

- Linear color transfer onto the style image, before style transfer.
- Style transfer only in the luminance channel.

Each method supplies perceptually-interesting effects but have blessings and disadvantages. The first method is clearly restrained through how well the colour transfer from the content image onto the style photograph works. The coloration distribution regularly cannot be matched perfectly, main to a mismatch between the colors of the output image and that of the content material photograph (Figure 4.14: Synthesis, using luminance-histogram matching before synthesis) The synthesis also replicates “content” structures from the van Gogh style scene, i.e., the pattern of reflections on the river appears as vertical yellow stripes of brushstrokes in the output.

In contrast, the second method preserves the colors of the content image perfectly. However, dependencies between the luminance and the color channels are lost in the

output image (Figure 4.13: Combination of synthesized luminance and source color channels.). This is particularly apparent for styles with prominent brushstrokes. In Figure 4.13 colors are no longer aligned to strokes. That means a single brushstroke can have multiple colors, which does not happen in real paintings. In comparison, when using full style transfer and color matching, the output image really consists of strokes which are blotches of paint, not just variations of light and dark.

“One potential advantage of the luminance-based method is that it reduces the dimensionality of the optimization problem for the neural synthesis[31].” The neural synthesis set of rules performs numerical optimization of the output photograph, and luminance-only synthesis has approx. 30% of its parameters. but, it's far doubtful that there's any sensible benefit in common GPU implementations.

Below marked figures shows different generated images after luminance-only synthesis for input images

Figure 4.10: Luminance channel (Y) of input photograph.

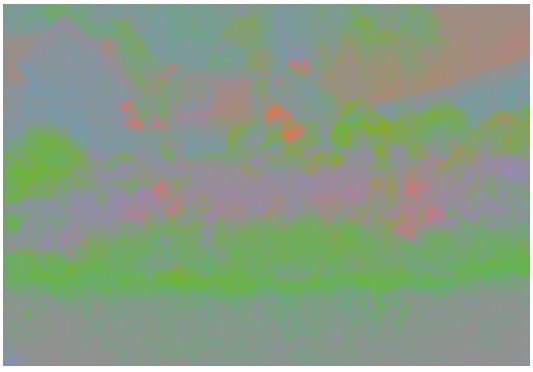


Figure 4.11: Color channels (I, Q) of input photograph

Figure 4.12: Style transfer result in luminance channel.

Figure 4.13: Combination of synthesized luminance and source color channels.

Figure 4.14: Synthesis, using luminance-histogram matching before synthesis

Figure 4.15: Combination of color and luminance channels, using luminance-histogram matching before synthesis.

CHAPTER 5 - CONCLUSION

We propose a more stable set of architectures for training generative adversarial networks and we give evidence that adversarial networks learn good representations of images for supervised learning and generative modeling. There are still some forms of model instability remaining - we noticed as models are trained longer they sometimes collapse a subset of filters to a single oscillating mode.

Further work is needed to tackle this from of instability. We think that extending this framework to other domains such as video (for frame prediction) and audio (pre-trained features for speech synthesis) should be very interesting. Further investigations into the properties of the learnt latent space would be interesting as well.

In future work, it would be interesting to explore how the two statistical models in here (color statistics vs. CNN activations) might be unified, and to explore more sophisticated color transfer and adjustment procedures .

REFERENCES

[1] I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio, "Maxout Networks," *30th Int. Conf. Mach. Learn. ICML 2013*, no. PART 3, pp. 2356–2364, Feb. 2013, Accessed: May 16, 2021. [Online]. Available: http://arxiv.org/abs/1302.4389.

[2] A. Coates and A. Y. Ng, "Learning feature representations with K-means," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 7700 LECTURE NO, pp. 561–580, 2012, doi: 10.1007/978-3-642-35289-8_30.

[3] P. V. Ca, L. T. Edu, I. Lajoie, Y. B. Ca, and P.-A. M. Ca, "Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion Pascal Vincent Hugo Larochelle Yoshua Bengio Pierre-Antoine Manzagol," 2010.

[4] J. Zhao, M. Mathieu, R. Goroshin, and Y. LeCun, "Stacked What-Where Auto-encoders," Jun. 2015, Accessed: May 16, 2021. [Online]. Available: http://arxiv.org/abs/1506.02351.

[5] A. Rasmus, H. Valpola, M. Honkala, M. Berglund, and T. Raiko, "Semi-supervised learning with Ladder networks," in *Advances in Neural Information Processing Systems*, Jul. 2015, vol. 2015-January, pp. 3546–3554, Accessed: May 16, 2021. [Online]. Available: https://arxiv.org/abs/1507.02672v2.

[6] H. Lee, R. Grosse, R. Ranganath, and A. Y. Ng, "Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations," 2009, pp. 1–8, doi: 10.1145/1553374.1553453.

[7] A. A. Efros and T. K. Leung, "Texture Synthesis by Non-parametric Sampling," 1999.

[8] W. T. Freeman, T. R. Jones, and E. C. Pasztor, "Example-Based Super-Resolution," no. April, pp. 56–65, 2002.

[9] J. Hays and A. A. Efros, "Scene completion using millions of photographs," *Commun. ACM*, vol. 51, no. 10, pp. 87–94, Oct. 2008, doi: 10.1145/1400181.1400202.

[10] J. Portilla and E. P. Simoncelli, "Parametric texture model based on joint statistics

of complex wavelet coefficients," *Int. J. Comput. Vis.*, vol. 40, no. 1, pp. 49–71, 2000, doi: 10.1023/A:1026553619983.

[11] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," Dec. 2014, Accessed: May 16, 2021. [Online]. Available: https://arxiv.org/abs/1312.6114v10.

[12] J. Sohl-Dickstein, E. A. Weiss, N. Maheswaranathan, and S. Ganguli, "Deep Unsupervised Learning using Nonequilibrium Thermodynamics," *32nd Int. Conf. Mach. Learn. ICML 2015*, vol. 3, pp. 2246–2255, Mar. 2015, Accessed: May 16, 2021. [Online]. Available: http://arxiv.org/abs/1503.03585.

[13] I. J. Goodfellow *et al.*, "Generative Adversarial Nets." Accessed: May 16, 2021. [Online]. Available: http://www.github.com/goodfeli/adversarial.

[14] E. Denton, S. Chintala, A. Szlam, and R. Fergus, "Deep Generative Image Models using a Laplacian Pyramid of Adversarial Networks," *Adv. Neural Inf. Process. Syst.*, vol. 2015-January, pp. 1486–1494, Jun. 2015, Accessed: May 16, 2021. [Online]. Available: http://arxiv.org/abs/1506.05751.

[15] K. Gregor, I. Danihelka, A. Graves, D. J. Rezende, and D. Wierstra, "DRAW: A recurrent neural network for image generation," in *32nd International Conference on Machine Learning, ICML 2015*, Feb. 2015, vol. 2, pp. 1462–1471, Accessed: May 16, 2021. [Online]. Available: https://www.youtube.

[16] A. Dosovitskiy, P. Fischer, J. T. Springenberg, M. Riedmiller, and T. Brox, "Discriminative Unsupervised Feature Learning with Exemplar Convolutional Neural Networks," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 38, no. 9, pp. 1734–1747, Jun. 2014, Accessed: May 16, 2021. [Online]. Available: http://arxiv.org/abs/1406.6909.

[17] M. D. Zeiler and R. Fergus, "LNCS 8689 - Visualizing and Understanding Convolutional Networks," 2014.

[18] "Google AI Blog: Inceptionism: Going Deeper into Neural Networks." https://ai.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html (accessed May 17, 2021).

[19] D. J. Heeger and J. R. Bergen, "Pyramid-based texture analysis/synthesis," in *Proceedings of the ACM SIGGRAPH Conference on Computer Graphics*, 1995, pp. 229–238, doi: 10.1145/218380.218446.

[20] L. A. Gatys, A. S. Ecker, and M. Bethge, “A Neural Algorithm of Artistic Style,” *J. Vis.*, vol. 16, no. 12, p. 326, Aug. 2015, Accessed: May 17, 2021. [Online]. Available: http://arxiv.org/abs/1508.06576.

[21] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” Sep. 2015, Accessed: May 17, 2021. [Online]. Available: http://www.robots.ox.ac.uk/.

[22] V. Deschaintre, M. Aittala, F. Durand, G. Drettakis, and A. Bousseau, “Single-Image SVBRDF Capture with a Rendering-Aware Deep Network,” *ACM Trans. Graph.*, vol. 37, no. 4, Oct. 2018, doi: 10.1145/3197517.3201378.

[23] D. Ulyanov, V. Lebedev, A. Vedaldi, and V. Lempitsky, “Texture Networks: Feed-forward Synthesis of Textures and Stylized Images,” *33rd Int. Conf. Mach. Learn. ICML 2016*, vol. 3, pp. 2027–2041, Mar. 2016, Accessed: May 17, 2021. [Online]. Available: http://arxiv.org/abs/1603.03417.

[24] J. Johnson, A. Alahi, and L. Fei-Fei, “Perceptual losses for real-time style transfer and super-resolution,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Mar. 2016, vol. 9906 LNCS, pp. 694–711, doi: 10.1007/978-3-319-46475-6_43.

[25] N. Srivastava, G. Hinton, A. Krizhevsky, and R. Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting,” 2014. Accessed: May 18, 2021. [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html.

[26] J. Deng *et al.*, “Imagenet: A large-scale hierarchical image database,” *CVPR*, 2009, Accessed: May 16, 2021. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.155.1729.

[27] A. Dosovitskiy, J. T. Springenberg, M. Tatarchenko, and T. Brox, “Learning to Generate Chairs, Tables and Cars with Convolutional Networks,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 4, pp. 692–705, Apr. 2017, doi: 10.1109/TPAMI.2016.2567384.

[28] A. Radford, L. Metz, and S. Chintala, “Unsupervised representation learning with deep convolutional generative adversarial networks,” Nov. 2016, Accessed: May 13, 2021. [Online]. Available: https://arxiv.org/abs/1511.06434v2.

[29] A. Coates and A. Y. Ng, “Selecting Receptive Fields in Deep Networks,” 2011.

[30] J. Bergstra, J. B. Ca, and Y. B. Ca, “Random Search for Hyper-Parameter Optimization Yoshua Bengio,” 2012. Accessed: May 13, 2021. [Online]. Available: http://scikit-learn.sourceforge.net.

[31] L. A. Gatys, U. Tübingen, M. Bethge, A. Hertzmann, A. Research, and E. Shechtman, “Preserving Color in Neural Artistic Style Transfer,” 2016.

APPENDIX

5.1 Code

5.1.1 Model Generation

```
import numpy as np
from keras.models import Sequential from keras.optimizers import Adam
from keras.layers import Dense, Conv2D, Flatten, Reshape, Conv2DTranspose
from keras.layers import LeakyReLU, Dropout
import matplotlib.pyplot as plt from keras.utils import plot_model
from keras.preprocessing.image import ImageDataGenerator from keras.models
import load_model

# define the discriminator model def define_D(in_shape=(128,128,3)):
    model = Sequential()
    model.add(Conv2D(64, (3,3), padding='same', input_shape=in_shape))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Flatten()) model.add(Dropout(0.4)) model.add(Dense(1,
    activation='sigmoid')) opt = Adam(lr=0.0002, beta_1=0.5)
    model.compile(loss='binary_crossentropy', optimizer=opt,
                  metrics=['accuracy'])
    return model

# define the generator model
def define_G(latent_dim):
    model = Sequential()
```

```
    # foundation for 16x16 image n_nodes = 256 * 16 * 16
        model.add(Dense(n_nodes, input_dim=latent_dim))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Reshape((16, 16, 256)))
    # upsample to 32x32
        model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
        model.add(LeakyReLU(alpha=0.2))
    # upsample to 64x64
        model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
        model.add(LeakyReLU(alpha=0.2))
    # upsamplde to 128x128
        model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Conv2D(3, (7,7), activation='tanh', padding='same'))
        return model

# define the combined generator and discriminator model, for updating the generator
def define_GAN(model_G, model_D):
# make weights in the discriminator not trainable model_D.trainable = False
        model = Sequential()
        model.add(model_G)
        model.add(model_D)
        opt = Adam(lr=0.0002, beta_1=0.5)
        model.compile(loss='binary_crossentropy', optimizer=opt)
        return model

from google.colab import drive drive.mount('/content/drive/')

def load_real_images():
        datagen = ImageDataGenerator(rescale=1./255)
        X = datagen.flow_from_directory('/content/drive/My Drive/Colab
```

```
        Notebooks/_images',  target_size= (128,128), batch_size=12500,
        class_mode='binary')
        data_list = []
        batch_index = 0
        while batch_index <= X.batch_index:
                data = X.next()
                data_list.append(data[0])
                batch_index += 1
        img_array = np.asarray(data_list)
        return img_array

def generate_real_images(dataset, n_samples):
        i = np.random.randint(0, dataset.shape[0], n_samples)
        X = dataset[i]
        y = np.ones((n_samples,1))
        return X, y

def generate_latent_points(latent_dim, n_samples):
        X = np.random.randn(latent_dim * n_samples)
        X = X.reshape(n_samples, latent_dim)
        return X

def generate_fake_images(model_G, latent_dim, n_samples):
        X_input = generate_latent_points(latent_dim, n_samples)
        X = model_G.predict(X_input)
        y = np.zeros((n_samples, 1))
        return X, y
def summarize_performance(epoch, model_G,
                                model_D, dataset, latent_dim, n_samples=100):
        model_G.save('/content/drive/My Drive/Colab Notebooks/model__'
        +str(epoch)+ '.h5')
```

```
X_real, y_real = generate_real_images(dataset, n_samples)
_, acc_real = model_D.evaluate(X_real, y_real, verbose=0)
x_fake, y_fake = generate_fake_images(model_G, latent_dim, n_samples)
_, acc_fake = model_D.evaluate(x_fake, y_fake, verbose=0)
print('Accuracy real: %.0f%%, fake: %.0f%%' % (acc_real*100, acc_fake*100))
def train_discriminator(model, dataset, n_iter=100, n_batch=256): half_batch =
int(n_batch/2)
# manually enumerate epochs for i in range(n_iter):
X_real, y_real = generate_real_images(dataset, half_batch)
_, real_acc = model.train_on_batch(X_real, y_real) X_fake, y_fake =
generate_fake_images(half_batch)
_, fake_acc = model.train_on_batch(X_fake, y_fake) print('%d real=%.0f%%
fake=%.0f%%' % (i+1, real_acc*100,
fake_acc*100))

def train_GAN(model_G, model_D, model_GAN, dataset, latent_dim, n_epochs=100,
n_batch=128):
bat_per_epo = int(dataset.shape[0] / n_batch) half_batch = int(n_batch / 2)
# manually enumerate epochs for i in range(n_epochs):
# enumerate batches over the training set for j in range(bat_per_epo):
X_real, y_real = generate_real_images(dataset,
half_batch)
X_fake, y_fake = generate_fake_images(model_G,
latent_dim, half_batch)
X, y = np.vstack((X_real, X_fake)), np.vstack((y_real, y_fake))
d_loss, _ = model_D.train_on_batch(X, y)
X_gan = generate_latent_points(latent_dim, n_batch) y_gan = np.ones((n_batch, 1))
g_loss = model_GAN.train_on_batch(X_gan, y_gan) print('%d, %d/%d, d=%.3f, g=%.3f
% (i+1, j+1,
bat_per_epo, d_loss, g_loss))
# evaluate the model performance if (i+1) % 10 == 0:
```

```
summarize_performance(i, model_G, model_D, dataset,
latent_dim)

latent_dim =100 model_D = define_D()
model_G =  define_G(latent_dim) model_GAN = define_GAN(model_G, model_D)
dataset=load_real_images()
train_GAN(model_G, model_D,model_GAN,dataset[0], latent_dim)

model = load_model('/content/drive/My Drive/Colab Notebooks/model_face_79.h5')

def generate_latent_points(latent_dim, n_samples): # generate points in the latent space
x_input = np.random.randn(latent_dim * n_samples) # reshape into a batch of inputs for
the network z_input = x_input.reshape(n_samples, latent_dim) return z_input

def plot_images(images, n):
# scales image values in the range of [0,1]
images = (images-images.min())/(images.max() - images.min()) for i in range(n):
# define subplot plt.subplot(1, n, 1 + i) # turn off axis plt.axis('off')
# plot raw pixel data plt.imshow(images[i, :, :])
plt.show()

pts = generate_latent_points(100, 30) # generate images
X = model.predict(pts) # plot the result plot_images(X, 4)
```

5.1.2 Image Generation

```
import torch
from torchvision import transforms , models
from PIL import Image
import matplotlib.pyplot as plt
import numpy as np
```

```
device = ("cuda" if torch.cuda.is_available() else "cpu") def
model_activations(input,model):
        layers = {
        '0' : 'conv1_1',
        '5' : 'conv2_1',
        '10': 'conv3_1',
        '19': 'conv4_1',
        '21': 'conv4_2',
        '28': 'conv5_1'
        }
        features = {} x = input
        x = x.unsqueeze(0)
        for name,layer in model._modules.items():
                x = layer(x)
                if name in layers:
                features[layers[name]] = x
        return features
transform = transforms.Compose([transforms.Resize(300),
transforms.ToTensor(), transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5))])

content = Image.open("content.png").convert("RGB") content =
transform(content).to(device) print("COntent shape => ", content.shape)
style = Image.open("style.jpg").convert("RGB") style = transform(style).to(device)

def imcnvt(image):
x = image.to("cpu").clone().detach().numpy().squeeze() x = x.transpose(1,2,0)
x = x*np.array((0.5,0.5,0.5)) + np.array((0.5,0.5,0.5)) return np.clip(x,0,1)

fig, (ax1,ax2) = plt.subplots(1,2)

ax1.imshow(imcnvt(content),label = "Content") ax2.imshow(imcnvt(style),label =
```

```
"Style")
plt.show()

def gram_matrix(imgfeature):
_,d,h,w = imgfeature.size() imgfeature = imgfeature.view(d,h*w)
gram_mat = torch.mm(imgfeature,imgfeature.t()) return gram_mat
target = content.clone().requires_grad_(True).to(device) #set device to cuda if available
print("device = ",device)


style_features = model_activations(style,model) content_features =
model_activations(content,model)

style_wt_meas = {"conv1_1" : 1.0,
"conv2_1" : 0.8,
"conv3_1" : 0.4,
"conv4_1" : 0.2,
"conv5_1" : 0.1}

style_grams = {layer:gram_matrix(style_features[layer]) for layer in style_features}

content_wt = 100 style_wt = 1e8

print_after = 500
epochs = 1000
optimizer = torch.optim.Adam([target],lr=0.007)

for i in range(1,epochs+1):
target_features = model_activations(target,model) content_loss =
torch.mean((content_features['conv4_2']-
target_features['conv4_2'])**2)
```

```
style_loss = 0
for layer in style_wt_meas: style_gram = style_grams[layer]
target_gram = target_features[layer]
_,d,w,h = target_gram.shape target_gram = gram_matrix(target_gram)

style_loss += (style_wt_meas[layer]*torch.mean((target_gram- style_gram)**2))/d*w*h

total_loss = content_wt*content_loss + style_wt*style_loss
if i%10==0:
print("epoch ",i," ", total_loss)

optimizer.zero_grad() total_loss.backward() optimizer.step()

if i%print_after == 0: plt.imshow(imcnvt(target),label="Epoch "+str(i)) plt.show()
plt.imsave(str(i)+'.png',imcnvt(target),format='png')
```

Vishrut_thakur_171309

ORIGINALITY REPORT

19%	19%	5%	9%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	docplayer.net Internet Source	8%
2	arxiv.org Internet Source	5%
3	towardsdatascience.com Internet Source	4%
4	www.arxiv-vanity.com Internet Source	1%
5	martychen920.blogspot.com Internet Source	1%
6	export.arxiv.org Internet Source	<1%

Exclude quotes	On	Exclude matches	< 14 words
Exclude bibliography	On		