MINOR PROJECT

Image Generation and Manipulation using Generative Adversarial Networks

Report submitted in partial fulfillment of the requirements for the award of

Degree of Bachelor of Technology

in

Software Engineering (SE)

Under the supervision of

Dr. Om Prakash Verma
(Professor, Computer Science and Engineering Department)

By:

Saksham Gupta (2K14/SE/076)

Satwik Kansal (2K14/SE/081)

Sparsh Bansal (2K14/SE/091)

To**:**

Department of Computer Science and Engineering

Delhi Technological University

(Formerly Delhi College of Engineering)

DECLARATION

I hereby certify that the work which is presented in the Minor Project entitled **“Image Generation using Generative Adversarial Network”** in fulfillment of the requirement for the award of the Degree of Bachelor of Technology and submitted to the Department of Computer Engineering, Delhi Technological University (Formerly Delhi College Of Engineering), New Delhi is an authentic record of my own, carried out during a period from August 2016 to November 2016, under the supervision of **Dr. Om Prakash Verma, Professor, CSE Department.**

The matter presented in this report has not been submitted by me for the award of any other degree of this or any other Institute/University.

Signature

SAKSHAM GUPTA	**2K14/SE/076**
SATWIK KANSAL	**2K14/SE/081**
SPARSH BANSAL	**2K14/SE/091**

ACKNOWLEDGEMENT

"The successful completion of any task would be incomplete without accomplishing the people who made it all possible and whose constant guidance and encouragement secured us the success."

First of all, we are grateful to the Almighty for establishing us to complete this minor project. We are grateful to **Dr. Om Prakash Verma, Professor** (Department of Computer Science and Engineering), Delhi Technological University (Formerly Delhi College of Engineering), New Delhi and all other faculty members of our department, for their astute guidance, constant encouragement and sincere support for this project work.

We owe a debt of gratitude to our guide, **Dr. Om Prakash Verma, Professor, CSE Department** for incorporating in us the idea of a creative Minor Project, helping us in undertaking this project and also for being there whenever we needed her assistance.

I also place on record, my sense of gratitude to one and all, who directly or indirectly have lent their helping hand in this venture. We feel proud and privileged in expressing my deep sense of gratitude to all those who have helped me in presenting this project.

Last but never the least, we thank our parents for always being with us, in every sense.

SUPERVISOR CERTIFICATE

This is to certify that **SAKSHAM GUPTA 2K14/SE/076, SATWIK KANSAL 2K14/SE/081 and SPARSH BANSAL 2K14/SE/091**, the bonafide students of Bachelor of Technology in Software Engineering of Delhi Technological University (Formerly Delhi College Of Engineering), New Delhi of 2013–2017 batch have completed their minor project entitled "**Image Generation using Generative Adversarial Network**" under the supervision of **Dr. Om Prakash Verma, Professor (Department of Computer Science and Engineering).** It is further certified that the work done in this dissertation is a result of candidate's own efforts.
I wish his/her all success in her life.

Date: 22nd November'2016

Dr. Om Prakash Verma
Professor
Computer Science & Engineering
Delhi Technological University
(Formerly Delhi College of Engineering)
Shahbad, Daulatpur, Bawana Road, Delhi – 110042

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Realistic image manipulation is challenging because it requires modifying the image appearance in a user-controlled way, while preserving the realism of the result. Unless the user has considerable, artistic skill, it is easy to "fall off" the manifold of natural images while editing. Generative Adversarial Networks can employed to generate photorealistic images by simultaneously training two models: A Generative Model *G* that captures the data distribution and a Discriminative Model *D* that estimates the probability that a sample came from the training data rather than *G*.

In this project, we try to learn the natural image manifold directly from data using a generative adversarial neural network We define a class of image editing operations, and constrain their output to lie on that learned manifold at all times. The model automatically adjusts the output keeping all edits as realistic as possible. All our manipulations are expressed in terms of constrained optimization and are applied in near-real time. We evaluate our algorithm on the task of realistic photo manipulation of shape and color.

Chapter 1

Introduction

1.1 Objective

The aim of the project is to **“Generate and Manipulate realistic images using Generative Adversarial Networks”**.

Description of Problem:

Today, visual communication is sadly one-sided. We all perceive information in the visual form (through photographs, paintings, sculpture), but only a chosen few are talented enough to effectively express themselves visually. This imbalance manifests itself even in the most mundane tasks. Consider an online shopping scenario: a user looking for shoes has found a pair that mostly suits her but she would like them to be a little taller, or wider, or in a different color. How can she communicate her preference to the shopping website? If the user is also an artist, then a few minutes with an image editing program will allow her to transform the shoe into what she wants, and then use image-based search to find it. However, for most of us, even a simple image manipulation in Photoshop presents insurmountable difficulties. One reason is the lack of “safety wheels” in image editing: any less-than-perfect edit immediately makes the image look completely unrealistic. To put another way, classic visual manipulation paradigm does not prevent the user from “falling off” the manifold of natural images.

Files/Datasets:

We experiment with Photo Collections from MIT Places dataset. The databases is scene-centric called Places, with 205 scene categories and 2.5 millions of images with a category label. The downloaded images are roughly centered but no further alignment has been performed.

1.2 Scope

We show the following varied applications based on our system:

- Manipulating an existing photo based on an underlying generative model to achieve a different look (shape and color).
- Generative transformation of one image to look more like another.
- Generate a new image from scratch based on user's scribbles and warping UI.

1.3 Project/Research Goals

This project aims at developing an accurate and realistic image generation and manipulation tool to assist in designing realistic images with minimum effort. The presented method can further be used for changing one image to look like the other, as well as generating novel imagery from scratch based on user's scribbles.

Chapter 2

Literature Survey

Ledig et al. (2016) demonstrates excellent single-image super resolution results that show the benefit of using a generative model trained to generate realistic samples from a multimodal distribution. The leftmost image is an original high-resolution image. It is then down sampled to make a low-resolution image, and different methods are used to attempt to recover the high-resolution image. The different methods include bicubic interpolation, SRResNet and SRGAN.

Brock et al. (2016) developed introspective adversarial networks (IAN). The user paints rough modifications to a photo, such as painting with black paint in an area where the user would like to add black hair, and IAN turns these rough paint strokes into photorealistic imagery matching the user's desires. Applications that enable a user to make realistic modifications to photo media are one of many reasons to study generative models that create images.

Isola et al. (2016) created a concept they called image to image translation, encompassing many kinds of transformations of an image: converting a satellite photo into a map, converting a sketch into a photorealistic image, etc. Because many of these conversion processes have multiple correct outputs for each input, it is necessary to use generative modeling to train the model correctly. In particular, Isola et al. (2016) use a GAN. Image to image translation provides many examples of how a creative algorithm designer can find several unanticipated uses for generative models. In the future, presumably many more such creative uses will be found.

Chapter 3

Fundamental Knowledge

3.1.1 Machine Learning

Machine learning is the subfield of computer science that, according to Arthur Samuel in 1959, gives "computers the ability to learn without being explicitly programmed." Evolved from the study of pattern recognition and computational learning theory in artificial intelligence, machine learning explores the study and construction of algorithms that can learn from and make predictions on data.

Machine learning algorithms are described as learning a target function (f) that best maps input variables (X) to an output variable (Y).

$$Y = f(X)$$

This is a general learning task where we would like to make predictions in the future (Y) given new examples of input variables (X).

We don't know what the function (f) looks like or it's form. If we did, we would use it directly and we would not need to learn it from data using machine learning algorithms. It is harder than you think. There is also error (e) that is independent of the input data (X).

$$Y = f(X) + e$$

This error might be error such as not having enough attributes to sufficiently characterize the best mapping from X to Y. This error is called irreducible error because no matter how good we get at estimating the target function (f), we cannot reduce this error.

This is to say, that the problem of learning a function from data is a difficult problem and this is the reason why the field of machine learning and machine learning algorithms exist.

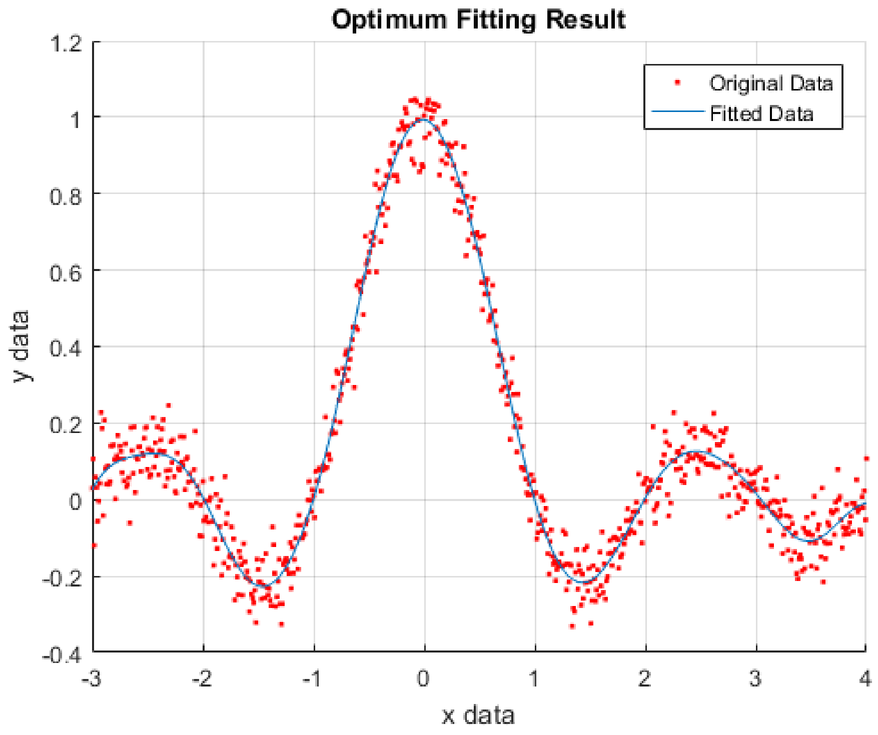

Fig 3.1 Plot illustrating the fitting of learned target function

3.2 Neural Networks

3.2.1 What is a neural network?

The basic idea behind a neural network is to simulate (copy in a simplified but reasonably faithful way) lots of densely interconnected brain cells inside a computer so you can get it to learn things, recognize patterns, and make decisions in a humanlike way. The amazing thing about a neural network is that you don't have to program it to learn explicitly: it learns all by itself, just like a brain!

But it isn't a brain. It's important to note that neural networks are (generally) software simulations: they're made by programming very ordinary computers, working in a very traditional fashion with their ordinary transistors and serially connected logic gates, to behave as though they're built from billions of highly interconnected brain cells working in parallel. No-one has yet attempted to build a computer by wiring up transistors in a densely parallel structure exactly like the human brain. In other words, a neural network differs from a human brain in exactly the same way that a computer model of the weather differs from real clouds, snowflakes, or sunshine. Computer simulations are just collections of algebraic variables and mathematical equations linking them together (in other words, numbers stored in boxes whose values are constantly changing). They mean nothing whatsoever to the computers they run inside—only to the people who program them.

3.2.2 What does a neural network consist of?

A typical neural network has anything from a few dozen to hundreds, thousands, or even millions of artificial neurons called units arranged in a series of layers, each of which connects to the layers on either side. Some of them, known as input units, are designed to receive various forms of information from the outside world that the network will attempt to learn about, recognize, or otherwise process. Other units sit on the opposite side of the network and signal how it responds to the information it's learned; those are known as output units. In between the input units and output units are one or more layers of hidden units, which, together, form the majority of the artificial brain. Most neural networks are fully connected, which means each hidden unit and each output unit is connected to every unit in the layers either side. The connections between one unit and another are represented by a number called a weight, which can be either positive (if one unit excites another) or negative (if one unit suppresses or inhibits another). The higher the weight, the more influence one unit has on another. (This corresponds to the way actual brain cells trigger one another across tiny gaps called synapses.)

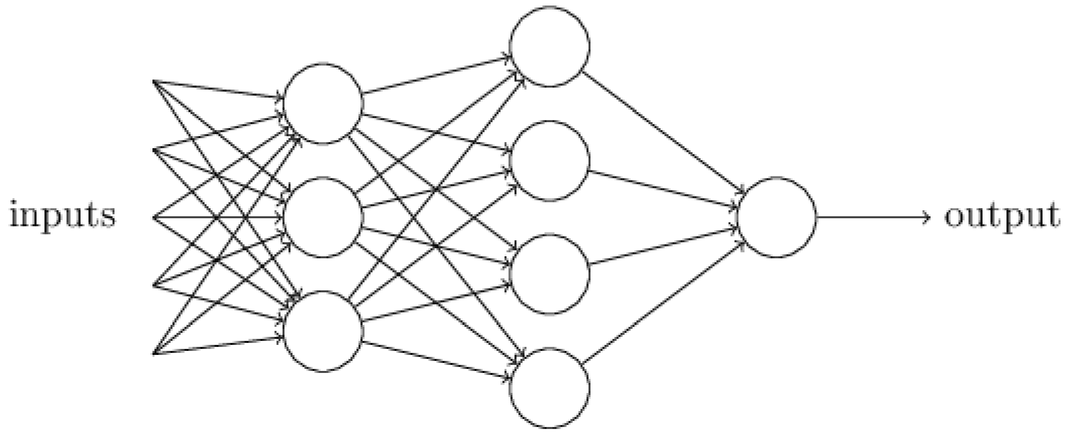

Fig 3.2
A conventional neural network architecture.

3.2.3 How does a neural network learn things?

Information flows through a neural network in two ways. When it's learning (being trained) or operating normally (after being trained), patterns of information are fed into the network via the input units, which trigger the layers of hidden units, and these in turn arrive at the output units. This common design is called a feedforward network. Not all units "fire" all the time. Each unit receives inputs from the units to its left, and the inputs are multiplied by the weights of the connections they travel along. Every unit adds up all the inputs it receives in this way and (in the simplest type of network) if the sum is more than a certain threshold value, the unit "fires" and triggers the units it's connected to (those on its right).

For a neural network to learn, there has to be an element of feedback involved—just as children learn by being told what they're doing right or wrong. In fact, we all use feedback, all the time. Think back to when you first learned to play a game like ten-pin bowling. As you picked up the heavy ball and rolled it down the alley, your brain watched how quickly the ball moved and the line it followed, and noted how close you came to knocking down the skittles. Next time it was your turn, you remembered what you'd done wrong before, modified your movements accordingly, and hopefully threw the ball a bit better. So you used feedback to compare the outcome you wanted with what actually happened, figured out the difference between the two, and used that to change what you did next time. The bigger the difference between the intended and actual outcome, the more radically you would have altered your moves.

Neural networks learn things in exactly the same way, typically by a feedback process called backpropagation (sometimes abbreviated as "backprop"). This involves comparing the output a network produces with the output it was meant to produce, and using the difference between them to modify the weights of the connections between the units in the network, working from the output units through the hidden units to the input units—going backward, in other words. In time, backpropagation causes the network to learn, reducing the difference between actual and intended output to the point where the two exactly coincide, so the network figures things out exactly as it should.

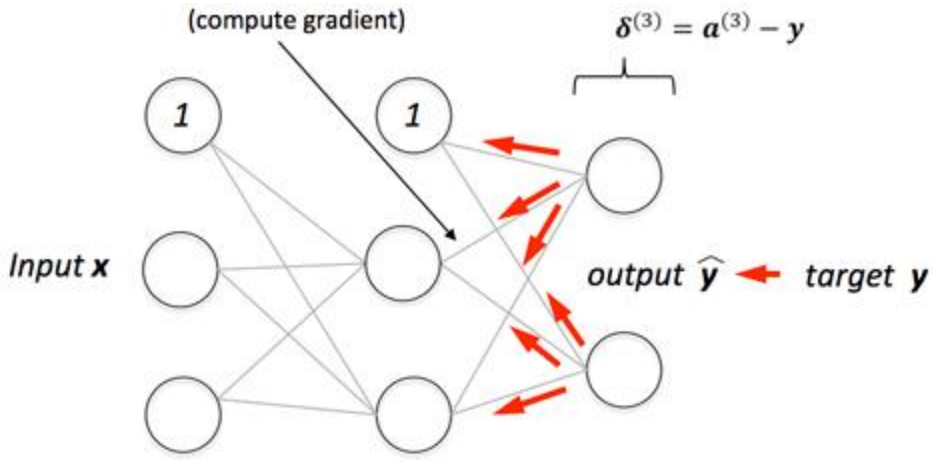

Fig 3.3 Backpropagation in neural networks

3.2.4 Deep Neural Networks

So, what's so Deep about Deep Neural Networks? Deep Neural Networks are basically Neural Networks with more than one Hidden layers. So, they look 'wider', rather than 'deeper'. There are few questions to be answered here –

If a single hidden layer network can approximate any function (UAT), why add multiple layers? This is one of the fundamental questions. Every hidden layer acts as a 'feature extractor.' If we have a just one hidden layers, two problems occur –

- The feature extraction capability of the network is very less, which means we have to provide suitable features to the network. This adds a feature extraction operation which is specific to that application. Therefore, the network, to some extent, loses its ability to learn a variety of functions, and cannot be called as 'automatic'.
- Even to learn the provided features, the number of nodes in the hidden layers grows exponentially, which causes arithmetic problems while learning.

To resolve this, we need the network to learn the features by itself. Therefore, we add multiple hidden layers each with less number of nodes. So, how well does this work? These Deep Neural Networks learned to play Atari games just by looking at the images from the screen.

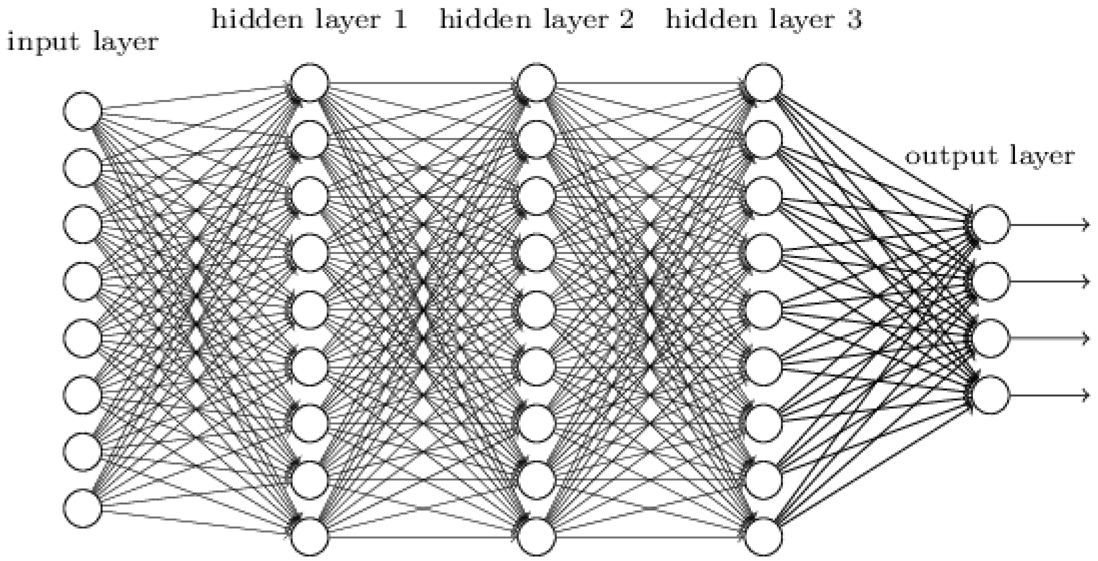

Fig 3.4 A deep neural network architecture of 5 layers.

3.2.5 **Generative Models**

In probability and statistics, a generative model is a model for randomly generating observable data values, typically given some hidden parameters. It specifies a joint probability distribution over observation and label sequences. Generative models are used in machine learning for either modeling data directly (i.e., modeling observations drawn from a probability density function), or as an intermediate step to forming a conditional probability density function. A conditional distribution can be formed from a generative model through Bayes' rule.

Generative models contrast with discriminative models, in that a generative model is a full probabilistic model of all variables, whereas a discriminative model provides a model only for the target variable(s) conditional on the observed variables. Thus a generative model can be used, for example, to simulate (i.e. generate) values of any variable in the model, whereas a discriminative model allows only sampling of the target variables conditional on the observed quantities. Despite the fact that discriminative models do not need to model the distribution of the observed variables, they cannot generally express more complex relationships between the observed and target variables. They don't necessarily perform better than generative models at classification and regression tasks. In modern applications the two classes are seen as complementary or as different views of the same procedure.

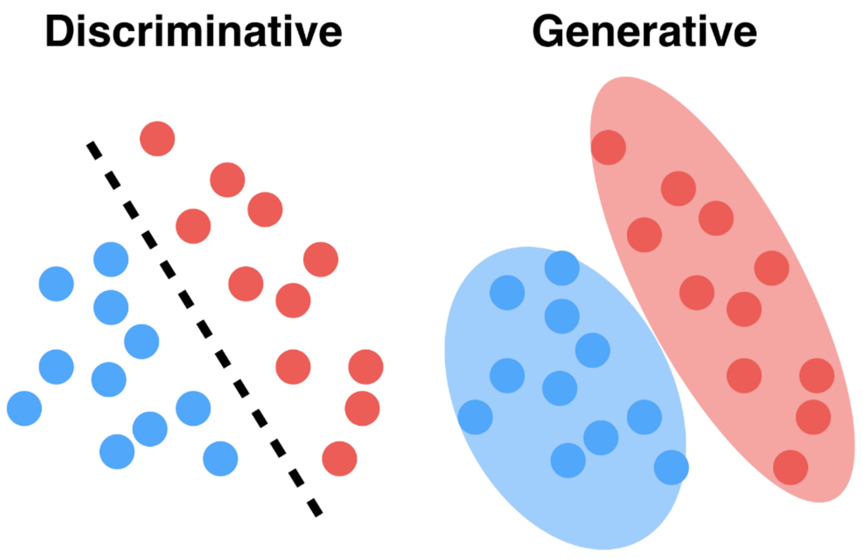

Fig 3.5 Target functions of generative and discriminative models.

Examples of generative models include:

- Gaussian mixture model and other types of mixture model
- Hidden Markov model
- Probabilistic context-free grammar
- Naive Bayes
- Averaged one-dependence estimators
- Latent Dirichlet allocation
- Restricted Boltzmann machine
- Generative adversarial networks

A generative algorithm models how the data was generated in order to categorize a signal. It asks the question: based on my generation assumptions, which category is most likely to generate this signal? A discriminative algorithm does not care about how the data was generated, it simply categorizes a given signal.

A generative algorithm models how the data was generated in order to categorize a signal. It asks the question: based on my generation assumptions, which category is most likely to generate this signal? A discriminative algorithm does not care about how the data was generated, it simply categorizes a given signal.

Suppose the input data is x and the set of labels for *x* is *y*. A generative model learns the joint probability distribution *p(x,y)* while a discriminative model learns the conditional probability distribution *p(y|x)* "probability of y given x".

So discriminative algorithms try to learn *p(y|x)* directly from the data and then try to classify data. On the other hand, generative algorithms try to learn *p(x,y)* which can be transformed into *p(y|x)* later to classify the data. One of the advantages of generative algorithms is that you can use *p(x,y)* to generate new data similar to existing data. On the other hand, discriminative algorithms generally give better performance in classification tasks.

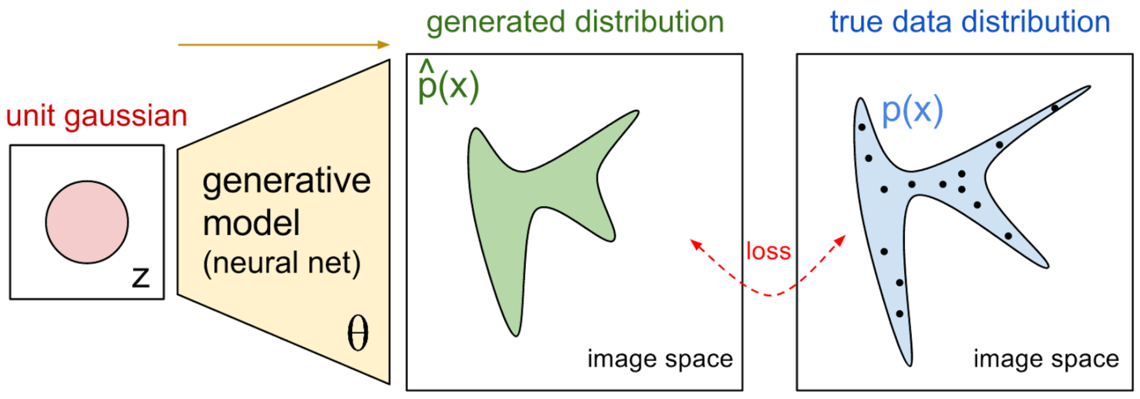

Fig 3.6 Learning true data distribution using Generative model.

3.2.6 **Generative Adversarial Networks**

Generative adversarial networks are a type of artificial intelligence algorithms used in unsupervised machine learning, implemented by a system of two neural networks competing against each other in a zero-sum game framework.

One network is generative and one is discriminative. Typically, the generative network is taught to map from a latent space to a particular data distribution of interest, and the discriminative network is simultaneously taught to discriminate between instances from the true data distribution and synthesized instances produced by the generator. The generative network's training objective is to increase the error rate of the discriminative network (i.e., "fool" the discriminator network by producing novel synthesized instances that appear to have come from the true data distribution). These models are used for computer vision tasks.

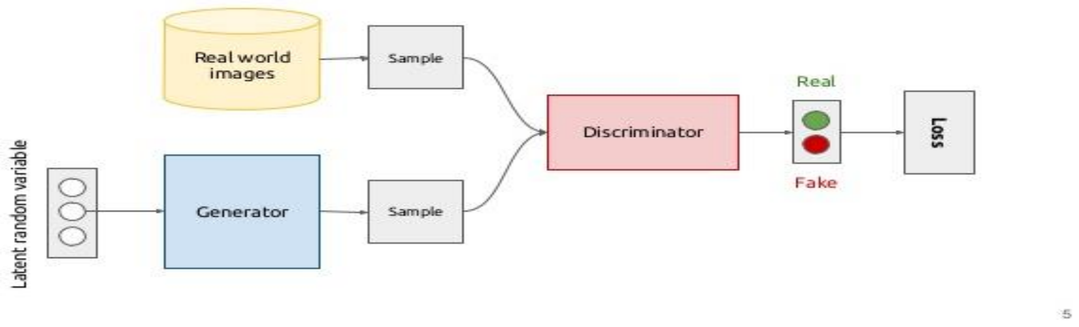

Fig 3.7 A conceptual architecture of Generative Adversarial Networks.

In practice, a particular dataset serves as the training data for the discriminator. Training the discriminator involves presenting the discriminator with samples from the dataset and samples synthesized by the generator, and backpropagating from a binary classification loss. In order to produce a sample, typically the generator is seeded with a randomized input that is sampled from a predefined latent space (e.g., a multivariate normal distribution). Training the generator involves back-propagating the negation of the binary classification loss of the discriminator. The generator adjusts its parameters so that the training data and generated data cannot be distinguished by the discriminator model. The goal is to find a setting of parameters that makes generated data look like the training data to the discriminator network. In practice, the generator is typically a deconvolutional neural network, and the discriminator is a convolutional neural network.

The adversarial modeling framework is most straightforward to apply when the models are both multilayer perceptrons. To learn the generator's distribution p_g over data x, we define a prior on input noise variables $p_z(z)$, then represent a mapping to data space as $G(z; \theta_g)$, where G is a differentiable function represented by a multilayer perceptron with parameters θ_g. We also define

a second multilayer perceptron *D(x; θd)* that outputs a single scalar. *D(x)* represents the probability that x came from the data rather than pg. We train D to maximize the probability of assigning the correct label to both training examples and samples from G. We simultaneously train *G* to minimize *log(1 − D(G(z)))*.
In other words, D and G play the following two-player minimax game with value function *V (G; D):*

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

3.2.7 Applications of Generative Adversarial Networks

There are several reasons to study generative models, including:

- Training and sampling from generative models is an excellent test of our ability to represent and manipulate high-dimensional probability distributions. High-dimensional probability distributions are important objects in a wide variety of applied math and engineering domains.

- Generative models can be incorporated into reinforcement learning in several ways. Generative models of time-series data can be used to simulate possible futures. A generative model used for planning can learn a conditional distribution over future states of the world, given the current state of the world and hypothetical actions an agent might take as input. The agent can query the model with different potential actions and choose actions that the model predicts are likely to yield a desired state of the world. Generative models can also be used to guide exploration by keeping track of how often different states have been visited or different actions have been attempted previously.

- Generative models can be trained with missing data and can provide predictions on inputs that are missing data. One particularly interesting case of missing data is semi-supervised learning, in which the labels for many or even most training examples are missing. Modern deep learning algorithms typically require extremely many labeled examples to be able to generalize well. Semi-supervised learning is one strategy for reducing the number of labels. The learning algorithm can improve its generalization by studying a large number of unlabeled examples which, which are usually easier to obtain.

- Many tasks intrinsically require realistic generation of samples from some distribution.

Chapter 4

Approach

Figure 4.1 illustrates the overview of our approach. Given a real photo, we first project it onto our approximation of the image manifold by finding the closest latent feature vector z of the GAN to the original image. Then, we present a real-time method for gradually and smoothly updating the latent vector z so that it generates a desired image that both satisfies the user's edits and stays close to the natural image manifold. Unfortunately, in this transformation the generative model usually loses some of the important low-level details of the input image. We therefore propose a dense correspondence method that estimates both per-pixel color and shape changes from the edits applied to the generative model. We then transfer these changes to the original photo using an edge-aware interpolation technique and produce the final manipulated result.

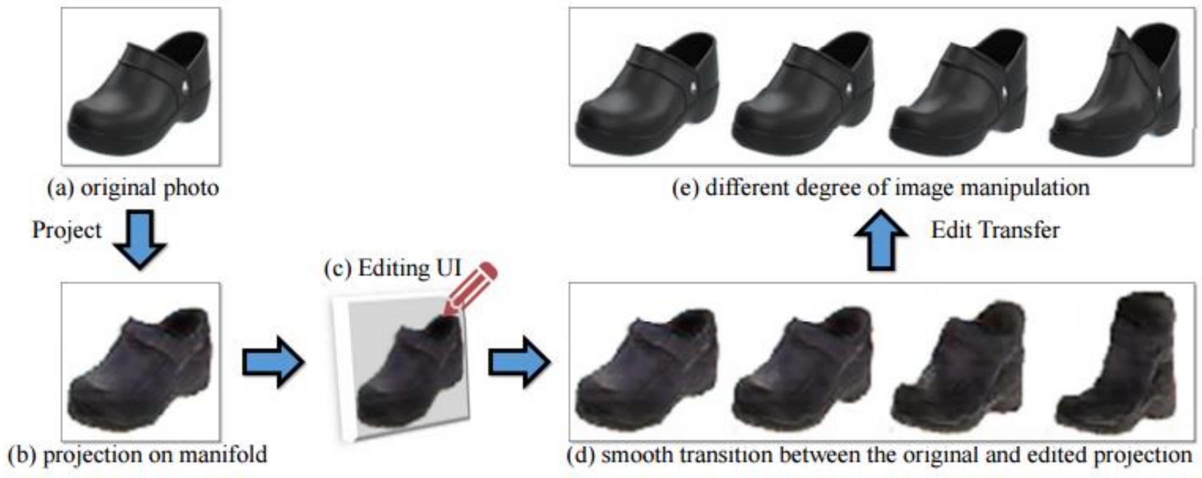
(a) original photo
Project
(b) projection on manifold
(c) Editing UI
(d) smooth transition between the original and edited projection
Edit Transfer
(e) different degree of image manipulation

Fig 4.1 a) The original image for training data
b) Representation of image projection on the image manifold.
c) Applying transformations to the image.
d) Interpolating the original and edited image on the image manifold and smoothening.
e) Generated images.

4.1 Projecting an image onto the manifold

A real photo x^R lies, by definition, on the ideal image manifold M. However for an approximate manifold M˜, our goal here is to find a generated image x∗ ∈ M˜ close to x R in some distance metric *L(x1, x2)* as

$$x^* = \arg\min_{x \in \tilde{\mathbb{M}}} \mathcal{L}(x, x^R).$$

For the GAN manifold M˜ we can rewrite the above equation as follows:

$$z^* = \arg\min_{z \in \tilde{\mathbb{Z}}} \mathcal{L}(G(z), x^R).$$

Our goal is to reconstruct the original photo x^R using the generative model *G* by minimizing the reconstruction error, where $L(x_1, x_2) = ||C(x_1) - C(x_2)||^2$ in some differentiable feature space *C*. If $C(x) = x$, then the reconstruction error is simply pixel-wise Euclidean error. Previous work suggests that using deep neural network activations leads to a reconstruction of perceptually meaningful details.

4.1.1 Projection via Optimization

As both the feature extractor C and the generative model G are differentiable, we can directly optimize the above objective using L-BFGS-B. However, the cascade of *C(G(z))* makes the problem highly non-convex, and as a result, the reconstruction quality strongly relies on a good initialization of *z*. We can start from multiple random initializations and output the solution with the minimal cost. However the number of random initializations required to obtain a stable reconstruction is prohibitively large (more than 100), which makes real-time processing impossible. We instead train a deep neural network to minimize the equation directly.

4.1.2 Projection via feed-forward network

We train a feedforward neural network *$P(x; \theta_P)$* that directly predicts the latent vector *z* from a *x*. The training objective for the predictive model P is written as follows:

$$\theta_P^* = \arg\min_{\theta_P} \sum_n \mathcal{L}(G(P(x_n^R; \theta_P)), x_n^R),$$

Where, x_n^Rdenotes the n-th image in the dataset. The architecture of the model P is equivalent to the discriminator D of the adversarial networks, and only varies in the final number of network outputs. Objective 3 is reminiscent of an auto-encoder pipeline, with a encoder *P* and decoder *G*. However, the decoder *G* is fixed throughout the training. While the optimization problem of the equation is exactly the same as the learning objective, the learning based approach often performs better and does not fall into local optima. We attribute this behavior to the regularity in the projection problem and the limited capacity of the network P. Projections of similar images will share similar network parameters and produce a similar result. In some sense the loss for one image provides information for many more images that share a similar appearance. However, the learned inversion is not always perfect, and can often be improved further by a few additional steps of optimization

4.1.3 A Hybrid Method

The hybrid method takes advantage of both approaches above. Given a real photo x^R, we first predict *$P(x^R; \theta_P)$* and then use it as the initialization for the optimization objective. So the predictive model we have trained serves as a fast bottom-up initialization method for a non-convex optimization problem.

4.1.4 Manipulating the latent vector

With the image x_0^R projected onto the manifold M˜ as $x_0 = G(z_0)$ via the projection methods just described, we can start modifying the image on that manifold. We update the initial projection x_0 by simultaneously matching the user intentions while staying on the manifold, close to the original image x_0. Each editing operation is formulated as a constraint $f_g(x) = v_g$ on a local part of the output image x. The editing operations g include color, shape and warping constraints. Given an initial projection x_0, we find a new image $x \in M$ close to x_0 trying to satisfy as many constraints as possible

$$x^* = \arg\min_{x \in \mathbb{M}} \Big\{ \underbrace{\sum_g \|f_g(x) - v_g\|^2}_{\text{data term}} + \underbrace{\lambda_s \cdot S(x, x_0)}_{\substack{\text{manifold}\\ \text{smoothness}}} \Big\},$$

Where, the data term measures deviation from the constraint and the smoothness term enforces moving in small steps on the manifold, so that the image content is not altered too much. We set $\lambda_s = 5$ in our experiments. The above equation simplifies to the following on the approximate GAN manifold M˜:

$$z^* = \arg\min_{z \in \mathbb{Z}} \Big\{ \underbrace{\sum_g \|f_g(G(z)) - v_g\|^2}_{\text{data term}} + \underbrace{\lambda_s \cdot \|z - z_0\|^2}_{\substack{\text{manifold}\\ \text{smoothness}}} + E_D \Big\}.$$

Here the last term $E_D = \lambda_D \cdot log(1 - D(G(z)))$ optionally captures the visual realism of the generated output as judged by the GAN discriminator D. This further pushes the image towards the manifold of natural images, and slightly improves the visual quality of the result. By default, we turn off this term to increase frame rates.

Gradient descent update: For most constraints the Equation is non-convex. We solve it using gradient descent, which allows us to provide the user with a real-time feedback as she manipulates the image. As a result, the objective evolves in real-time as well. For computational reasons, we only perform a few gradient descent updates after changing the constraints v_g. Once the final result $G(z_1)$ is computed, a user can see the interpolation sequence between the initial point z_0 and z_1, and select any intermediate result as the new starting point. Please see supplemental video for more details. While this editing framework allows us to modify any generated image on the approximate natural image manifold M˜, it does not directly provide us a way to modify the original high resolution image x_0^R.

4.2 Edit Transfer

4.2.1 Motion & Color Flow Algorithm

Give the original photo x_0^R (e.g. a black shoe) and its projection on the manifold G(z$_0$), and a user modification G(z$_1$) by our method (e.g. the generated red shoe). The generated image G(z$_1$) captures the roughly change we want, albeit the quality is degraded w.r.t the original image.

Can we instead adjust the original photo and produce a more photo-realistic result x_1^R that exhibits the changes in the generated image? A straightforward way is to transfer directly the pixel changes (i.e. x_1^R = x_0^R + (G(z$_1$)−G(z$_0$)). We have tried this approach and it introduces new artifacts due to the misalignment of the two images. To address this issue, we develop a dense correspondence algorithm to estimate both the geometric and color changes induced by the editing process.

Specifically, given two generated images G(z$_0$) and G(z$_1$), we can generate any number of intermediate frames $[G(\left(1-\frac{t}{N}\right).z_0+\frac{t}{N}.z_1)]_{t=0}^{N}$, where consecutive frames only exhibit minor visual variations.

We then estimate the color and geometric changes by generalizing the brightness constancy assumption in traditional optical flow methods. This results in the following motion and color flow objective:

$$\iint \underbrace{\|I(x,y,t)-A\cdot I(x+u,y+v,t+1)\|^2}_{\text{data term}}+\underbrace{\sigma_s(\|\nabla u\|^2+\|\nabla v\|^2)}_{\text{spatial reg}}+\underbrace{\sigma_c\|\nabla A\|^2}_{\text{color reg}}dxdy,$$

Where, *I(x, y, t)* denotes the RGB values *(r, g, b, 1)^T* of pixel *(x, y)* in the generated image

$G\left(\left(1-\frac{t}{N}\right).z_0+\frac{t}{N}.z_1\right).(u,v)$ is the flow vector with respect to the change of t, and A denotes a 3 × 4 color affine transformation matrix. The data term relaxes the color constancy assumption by introducing a locally affine color transfer model A while the spatial and color regularization terms encourage smoothness in both the motion and color change. We solve the objective by iteratively estimating the flow (u, v) using a traditional optical flow algorithm, and computing the color change A by solving a system of linear equations. We iterate 3 times. We produce 8 intermediate frames (i.e. N = 7).

We estimate the changes between nearby frames, and concatenate these changes frame by frame to obtain long-range changes between any two frames along the interpolation sequence z$_0$ → z$_1$.

Transfer edits to the original photo: After estimating the color and shape changes in the generated image sequence, we apply them to the original photo and produce an interesting transition sequence of photo-realistic images as shown in Figure 5. As the resolution of the flow and color fields are limited to the resolution of the generated image (i.e. 64 × 64), we up sample those edits using a guided image filter.

4.2.2 Editing Constraints

Our system provides three constraints to edit the photo in different aspects: coloring, sketching and warping. All constraints are expressed as brush tools. In the following, we explain the usage of each brush, and the corresponding constraints.

- Coloring brush: The coloring brush allows the user to change the color of a specific region. The user selects a color from a palette and can adjust the brush size. For each pixel marked with this brush we constrain the color $f_g(I) = I_p = v_g$ of a pixel p to the selected values v_g.
- Sketching brush: The sketching brush allows the user to outline the shape or add fine details. We constrain $f_g(I) = HOG(I)_p$ a differentiable HOG descriptor at a certain location p in the image to be close to the user stroke (i.e. $v_g = HOG(stroke)_p$). We chose the HOG feature extractor because it is binned, which makes it robust to sketching inaccuracies.

Chapter 5

Implementation

5.1 Training Generative Adversarial Networks

Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyper parameter.

```
for number of training iterations do
    for k steps do
    • Sample minibatch of m noise samples $\{z^{(1)} \dots z^{(m)}\}$ from noise prior $p_g(z)$.
    • Sample minibatch of m examples $\{x^{(1)} \dots x^{(m)}\}$ from data generating
      distribution $p_{data}(x)$.
    • Update the discriminator by ascending its stochastic gradient:
          $\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[ \log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$
    end for

  • Sample mini-batch of m noise samples $\{z^{(1)} \dots z^{(m)}\}$ from noise prior $p_g(z)$.
  • Update the generator by descending its stochastic gradient:
          $\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$
  end for
```

The gradient-based updates can use any standard gradient-based learning rule.

5.2 Network Architecture

Network architecture: We follow the same architecture of deep convolutional generative adversarial networks (DCGAN) . DCGAN mainly builds on multiple convolution, deconvolution and ReLU layers, and eases the min-max training via batch normalization. We train the generator G to produce a 64 × 64 × 3 image given a 100-dimensional random vector. Notice that our method can also use other generative models (e.g. variational auto-encoder or future improvements in this area) to approximate the natural image manifold.

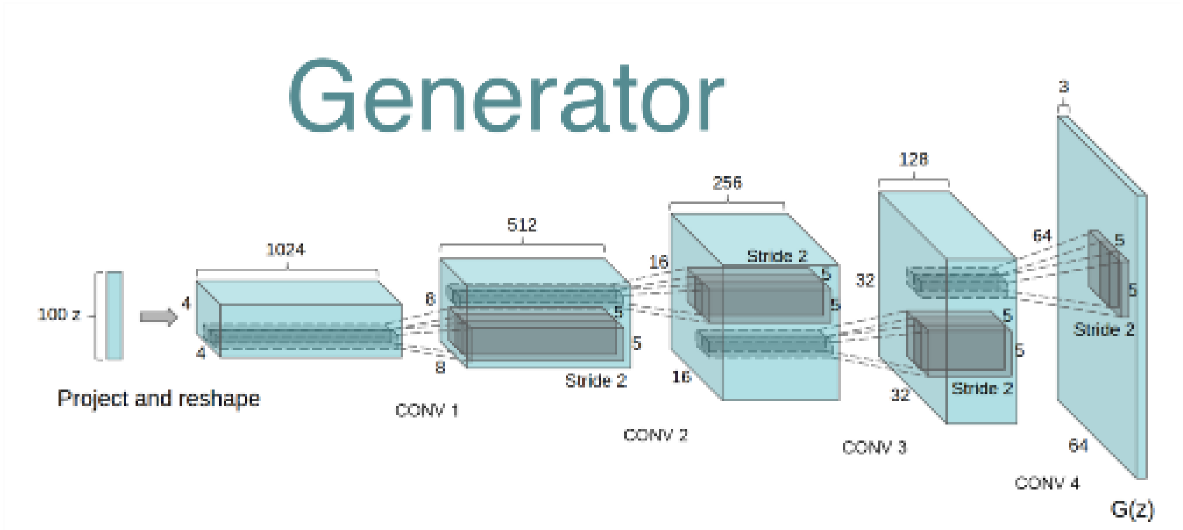

Fig 5.1 The Generator architecture.

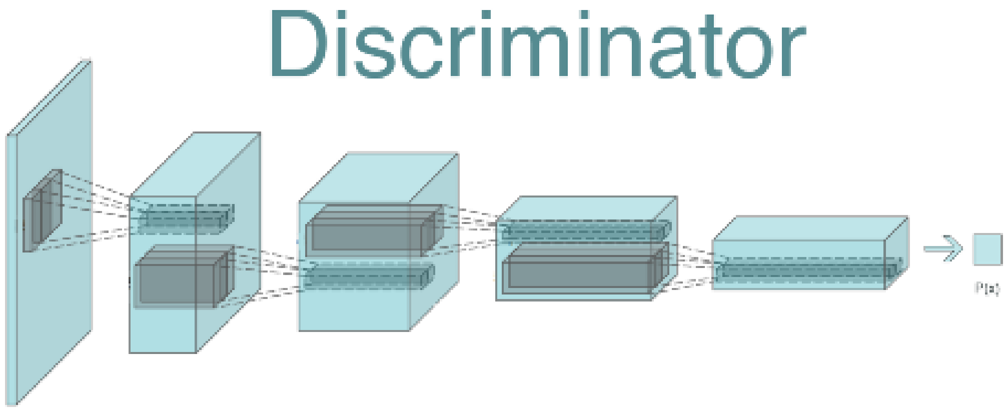

Fig 5.2 The Discriminator architecture.

Chapter 6

Results and Analysis

Class-specific model: So far, we have trained the generative model on a particular class of images. As a comparison, we train a cross-class model on three datasets altogether (i.e. shoes, handbags, and shirts), and observe that the model achieves worse reconstruction error compared to class-specific models (by ~ 10%). We also have tried to use a class-specific model to reconstruct images from a different class. The mean cross-category reconstruction errors are much worse: shoes model used for shoes: 0.140 vs. shoes model for handbags: 0.398, and for shirts: 0.451. However, we expect a model trained on many categories (e.g. 1, 000) to generalize better to novel objects. Perception study: We perform a small perception study to compare the photo realism of four types of images: real photos, generated samples produced by GAN, our method (shape only), and our method (shape and color).

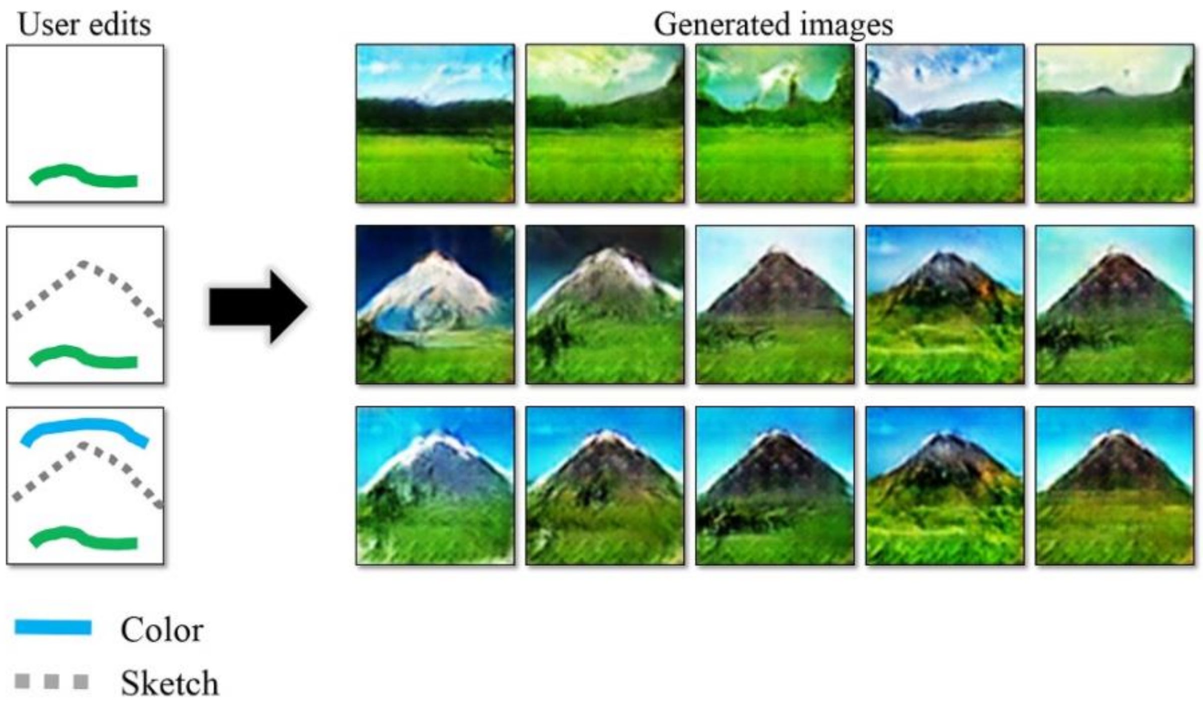

Fig 6.1 Demonstrating the process of generating images from simple coloring and sketching

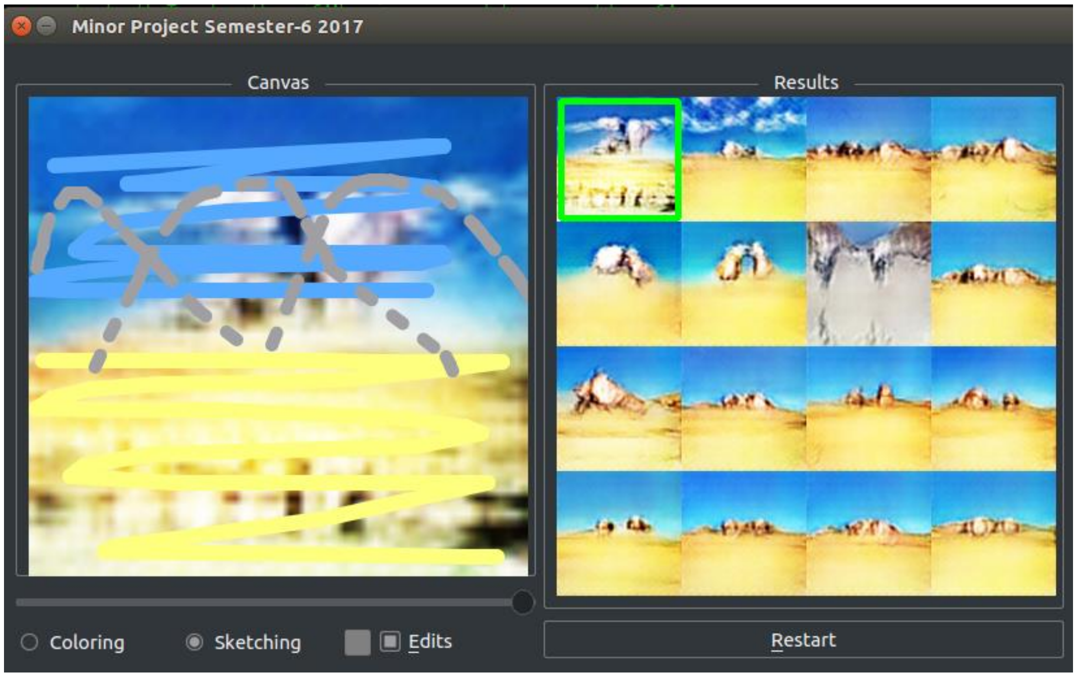

Fig 6.2 Generating image of a desert area.

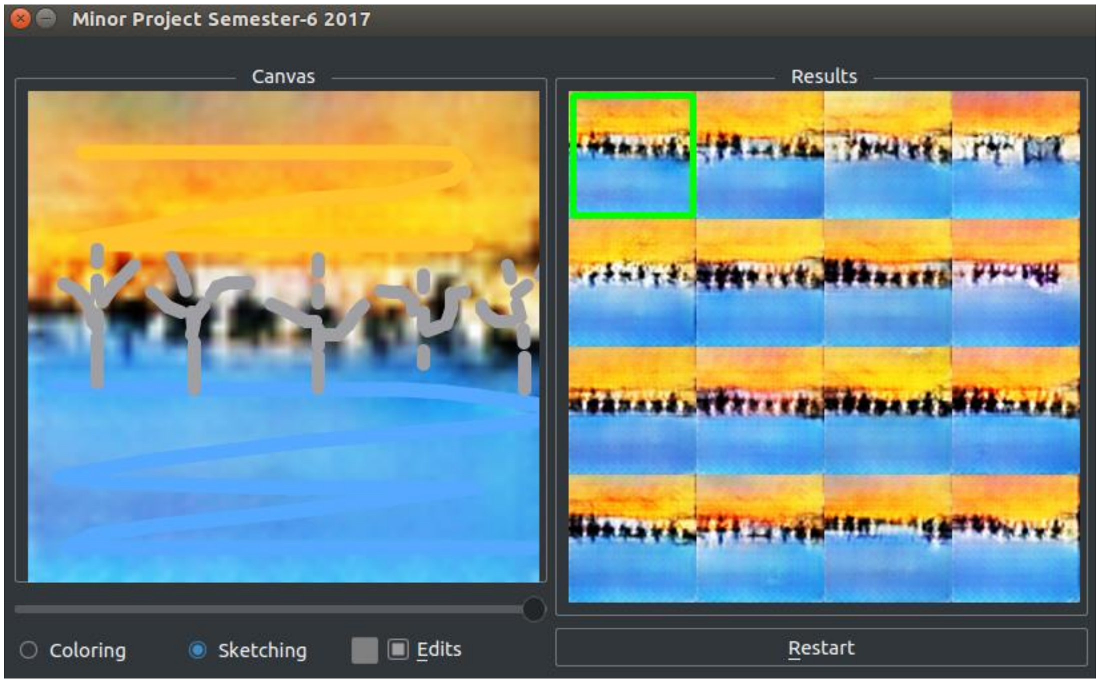

Fig 6.3 Generating image of distant trees under an orange sky.

As we can see the results from our Generative Adversarial Network look promising and perceptually appealing.

Chapter 7

Conclusion

7.1 Conclusion

We presented a step towards image editing with a direct constraint to stay close to the manifold of real images. We approximate this manifold using the state-of the-art in deep generative models (DCGAN). We show how to make interactive edits to the generated images and transfer the resulting changes in shape and color back to the original image. Thus, the quality of the generated results (low resolution, missing texture and details) and the types of data DCGAN is applicable to (works well on structured datasets such as product images and worse on more general imagery), limits how far we can get with this editing approach. However our method is not tied to a particular generative method and will improve with the advancement of this field. Our current editing brush tools allow rough changes in color and shape but not texture and more complex structure changes. We leave these for future work.

7.2 Limitations

Non-Convergence

The largest problem facing GANs that researchers should try to resolve is the issue of non-convergence. Most deep models are trained using an optimization algorithm that seeks out a low value of a cost function. While many problems can interfere with optimization, optimization algorithms usually make reliable downhill progress. GANs require finding the equilibrium to a game with two players. Even if each player successfully moves downhill on that player's update, the same update might move the other player uphill. Sometimes the two players eventually reach an equilibrium, but in other scenarios they repeatedly undo each others' progress without arriving anywhere useful. This is a general problem with games not unique to GANs, so a general solution to this problem would have wide-reaching applications.

Evaluation of Generative Models

Another highly important research area related to GANs is that it is not clear how to quantitatively evaluate generative models. Models that obtain good likelihood can generate bad samples, and models that generate good samples can have poor likelihood. There is no clearly justified way to quantitatively score samples. GANs are somewhat harder to evaluate than other generative models because it can be difficult to estimate the likelihood for GANs (but it is possible—see Wu et al. (2016)). Theis et al. (2015) describe many of the difficulties with evaluating generative models.

References

1. Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y.: Generative adversarial nets. In: NIPS. (2014) 2672–2680

2. Radford, A., Metz, L., Chintala, S.: Unsupervised representation learning with deep convolutional generative adversarial networks. ICLR (2016)

3. Kingma, D.P., Welling, M.: Auto-encoding variational bayes. ICLR (2014)

4. Denton, E.L., Chintala, S., Fergus, R., et al.: Deep generative image models using a laplacian pyramid of adversarial networks. In: NIPS. (2015) 1486–1494

5. Dosovitskiy, A., Brox, T.: Generating images with perceptual similarity metrics based on deep networks. arXiv preprint arXiv:1602.02644 (2016)

6. Reinhard, E., Ashikhmin, M., Gooch, B., Shirley, P.: Color transfer between images. IEEE Comput. Graph. Appl. (September 2001 2001)

7. Ledig, C., Theis, L., Huszar, F., Caballero, J., Aitken, A. P., Tejani, A., Totz, J., Wang, Z., and Shi, W. (2016). Photo-realistic single image super-resolution using a generative adversarial network. CoRR, abs/1609.04802.

8. Brock, A., Lim, T., Ritchie, J. M., and Weston, N. (2016). Neural photo editing with introspective adversarial networks. CoRR, abs/1609.07093

9. Isola, P., Zhu, J.-Y., Zhou, T., and Efros, A. A. (2016). Image-to-image translation with conditional adversarial networks. arXiv preprint arXiv:1611.07004 .

Training Module

```
from __future__ import print_function
import sys
sys.path.append('..')
import os
import json
from time import time
import numpy as np
from tqdm import tqdm
import theano
import theano.tensor as T
import train_dcgan_config
from lib import updates
from lib import utils
from lib.rng import py_rng, np_rng
from lib import costs
import train_dcgan_utils
from lib.theano_utils import floatX, sharedX
import load
from lib import image_save
import argparse

# set arguments and parameters
parser = argparse.ArgumentParser('Train DCGAN model')
parser.add_argument('--model_name', dest='model_name',
help='model name', default='shoes_64', type=str)
parser.add_argument('--ext', dest='ext', help='experiment
name=model_name+ext', default='', type=str)
parser.add_argument('--data_file', dest='data_file', help='the
file that stores the hdf5 data', type=str, default=None)
parser.add_argument('--cache_dir', dest='cache_dir', help='cache
directory that stores models, samples and webpages', type=str,
default=None)
```

```
parser.add_argument('--batch_size', dest='batch_size', help='the
number of examples in each batch', type=int, default=128)

parser.add_argument('--update_k', dest='update_k', help='the
number of discrim updates for each gen update', type=int,
default=2)
parser.add_argument('--save_freq', dest='save_freq', help='save
a model every save_freq epochs', type=int, default=1)
parser.add_argument('--lr', dest='lr', help='learning rate',
type=float, default=0.0002)
parser.add_argument('--weight_decay', dest='weight_decay',
help='l2 weight decay', type=float, default=1e-5)
parser.add_argument('--b1', dest='b1', help='momentum term of
adam', type=float, default=0.5)
args = parser.parse_args()


if not args.data_file:
   args.data_file = '../datasets/%s.hdf5' % args.model_name

n_vis = 196
npx, n_layers, n_f, nc, nz, niter, niter_decay =
getattr(train_dcgan_config, args.model_name)()
expr_name = args.model_name + args.ext

if not args.cache_dir:
   args.cache_dir = './cache/%s/' % expr_name

for arg in vars(args):
   print('[%s] =' % arg, getattr(args, arg))

# create directories
sample_dir = os.path.join(args.cache_dir, 'samples')
model_dir = os.path.join(args.cache_dir, 'models')
log_dir = os.path.join(args.cache_dir, 'log')
web_dir = os.path.join(args.cache_dir, 'web_dcgan')
html = image_save.ImageSave(web_dir, expr_name, append=True)
utils.mkdirs([sample_dir, model_dir, log_dir, web_dir])

# load data from hdf5 file
tr_data, te_data, tr_stream, te_stream, ntrain, ntest =
load.load_imgs(ntrain=None, ntest=None,
batch_size=args.batch_size,data_file=args.data_file)
te_handle = te_data.open()
test_x, = te_data.get_data(te_handle, slice(0, ntest))

# generate real samples and test transform/inverse_transform
test_x = train_dcgan_utils.transform(test_x, nc=nc)
vis_idxs = py_rng.sample(np.arange(len(test_x)), n_vis)
vaX_vis = train_dcgan_utils.inverse_transform(test_x[vis_idxs],
npx=npx, nc=nc)
# st()
n_grid = int(np.sqrt(n_vis))
grid_real = utils.grid_vis((vaX_vis*255.0).astype(np.uint8),
n_grid, n_grid)
train_dcgan_utils.save_image(grid_real, os.path.join(sample_dir,
'real_samples.png'))


# define DCGAN model
disc_params = train_dcgan_utils.init_disc_params(n_f=n_f,
n_layers=n_layers, nc=nc)
gen_params = train_dcgan_utils.init_gen_params(nz=nz, n_f=n_f,
n_layers=n_layers, nc=nc)
x = T.tensor4()
z = T.matrix()

gx = train_dcgan_utils.gen(z, gen_params, n_layers=n_layers,
n_f=n_f, nc=nc)
p_real = train_dcgan_utils.discrim(x, disc_params,
n_layers=n_layers)
p_gen = train_dcgan_utils.discrim(gx, disc_params,
n_layers=n_layers)

d_cost_real = costs.bce(p_real, T.ones(p_real.shape))
d_cost_gen = costs.bce(p_gen, T.zeros(p_gen.shape))
g_cost_d = costs.bce(p_gen, T.ones(p_gen.shape))

d_cost = d_cost_real + d_cost_gen
g_cost = g_cost_d

cost = [g_cost, d_cost, g_cost_d, d_cost_real, d_cost_gen]
```

```
lrt = sharedX(args.lr)
d_updater = updates.Adam(lr=lrt, b1=args.b1,
regularizer=updates.Regularizer(l2=args.weight_decay))
g_updater = updates.Adam(lr=lrt, b1=args.b1,
regularizer=updates.Regularizer(l2=args.weight_decay))
d_updates = d_updater(disc_params, d_cost)
g_updates = g_updater(gen_params, g_cost)
updates = d_updates + g_updates

print('COMPILING')
t = time()
_train_g = theano.function([x, z], cost, updates=g_updates)
_train_d = theano.function([x, z], cost, updates=d_updates)
_gen = theano.function([z], gx)
print('%.2f seconds to compile theano functions' % (time() - t))

# test z samples
sample_zmb = floatX(np_rng.uniform(-1., 1., size=(n_vis, nz)))


f_log = open('%s/training_log.ndjson' % log_dir, 'wb')
log_fields = ['n_epochs', 'n_updates', 'n_examples',
'n_seconds', 'g_cost', 'd_cost',]

# initialization
n_updates = 0
n_epochs = 0
n_examples = 0
t = time()

for epoch in range(niter+niter_decay):
   for imb, in tqdm(tr_stream.get_epoch_iterator(), total=ntrain
/ args.batch_size):
       imb = train_dcgan_utils.transform(imb, nc=nc)
       zmb = floatX(np_rng.uniform(-1., 1., size=(len(imb),
nz)))
       if n_updates % args.update_k == 0:
           cost = _train_g(imb, zmb)
       else:
           cost = _train_d(imb, zmb)
```

```
        n_updates += 1
        n_examples += len(imb)

    g_cost = float(cost[0])
    d_cost = float(cost[1])
    # print logging information
    log = [n_epochs, n_updates, n_examples, time() - t,  g_cost,
d_cost]
    print('epoch %.0f: G_cost %.4f, D_cost %.4f' %
(epoch,  g_cost, d_cost))
    f_log.write(json.dumps(dict(zip(log_fields, log))) + '\n')
    f_log.flush()

    n_epochs += 1

    # generate samples and write webpage
    samples = np.asarray(_gen(sample_zmb))
    samples_t = train_dcgan_utils.inverse_transform(samples,
npx=npx, nc=nc)
    grid_vis = utils.grid_vis(samples_t, n_grid, n_grid)
    grid_vis_i = (grid_vis*255.0).astype(np.uint8)
    train_dcgan_utils.save_image(grid_vis_i,
os.path.join(sample_dir, 'gen_%5.5d.png'%n_epochs))
    html.save_image([grid_vis_i], [''], header='epoch_%3.3d' %
n_epochs, width=grid_vis.shape[1], cvt=True)
    html.save()

    # save models
    if n_epochs > niter:
        lrt.set_value(floatX(lrt.get_value() - args.lr /
niter_decay))
    if n_epochs % args.save_freq == 0:
        train_dcgan_utils.save_model(disc_params,
'%s/disc_params_%3.3d' % (model_dir, n_epochs))
        train_dcgan_utils.save_model(gen_params,
'%s/gen_params_%3.3d' % (model_dir, n_epochs))
    train_dcgan_utils.save_model(disc_params, '%s/disc_params' %
model_dir)
    train_dcgan_utils.save_model(gen_params, '%s/gen_params' %
model_dir)
```