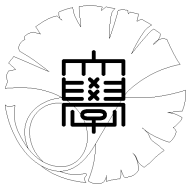
Master's Thesis

Generative Adversarial Network for Understanding Latent Space (潜在空間を理解可能な敵対的生成ネットワーク)

Supervisor 上條 俊介 准教授



東京大学大学院
情報理工学系研究科
電子情報学専攻

Student Number, Name 48–166466 劉 永劼

Date January 31, 2018

Abstract

The contribution of this thesis is to explore a latent space understandable Generative Adversarial Network (GAN), we name it as Self-excited Generative Adversarial Network (SelfExGAN). A novel self-excited structure based on adversarial learning. In this thesis, we take a review of deep learning and many extended versions of GANs. Compared with the conventional GANs, SelfExGAN consists of three components, which are encoder (E), generator (G), and discriminator (D). Different from other proposals which directly apply reconstruction loss between encoder and generator, SelfExGAN introduces Nash equilibrium between these three parts in order to discover the correspondence between latent inputs space and training data space spontaneously. The most attractive point of SelfExGAN is that it can use the learned correspondence to guide G to generate homomorphic samples given different latent inputs in an unsupervised learning manner. SelfExGAN learns how to generate new samples, but also determines the correspondence between the latent inputs space and the training data space.

Our proposed model has various applications, for example, generating training data with the label for supervised learning, evaluation on the similarity of two samples, and improving the divergent creation for design. In Chapter 1, we give an introduction of our work. In Chapter 2, we take a review of deep learning and Generative Adversarial Network. In Chapter 3, we describe our proposed model: a latent space understandable GAN, SelfExGAN, as well as with the theoretical results. Its implementation and application will be introduced in Chapter 4 which show that SelfExGAN performs well in generating homomorphic images. We present conclusions and future works in Chapter 5.

Contents

Chapter 1 Introduction 1
1.1 Basic Concepts 1
1.2 Famous DNN Architectures 3
1.3 Thesis Structure 9

Chapter 2 Generative Adversarial Networks 10
2.1 Layers 10
2.2 Activation Functions 13
2.3 Training 16
2.4 Generative Adversarial Networks 28

Chapter 3 Lantent Space Understandable Generative Adversarial Netork 32
3.1 Introduction and Intuition 32
3.2 Related Works 34
3.3 Self-excited Generative Adversarial Network 37

Chapter 4 Experiments and Applications 46
4.1 Experiments 46
4.2 Applications 63

Chapter 5 Conclusion and Future Works 67

Thanks 68

Publications 69

Reference 70

A Derivation for Equation (4.5) 74

List of Figures

1.1 Model of biological neuron. Image is taken from Wikipedia. 1
1.2 Model of an artificial neuron 2
1.3 Brain Structure [1]. Explain how human's visual system works. 2
1.4 A Multilayer perceptron network consists of three layers 3
1.5 Architecture of LeNet [2]. 4
1.6 Architecture of AlexNet [3]. 5
1.7 Visualizations of layer 1 and layer 2 [4]. Each layer illustrated 2 pictures, one which shows the filters themselves and that shows what part of the image are most strongly activated by the given filter. For example, in the space labled layer 2, we have representation of the 16 different filters (on the left) 6
1.8 More visualization on layer 3,4,5 [4]. 6
1.9 The 6 different architectures of VGGNet in original paper [5]. 7
1.10 Full inception module architecture [6]. 8
1.11 A Residual Block [7]. 9

2.1 A cartoon drawing of a biological neuron (left), and its mathematical model (right). Image is taken from Wikipedia. 11
2.2 Convolution with 2×2 filter. Image is taken from Wikipedia. 12
2.3 Max pooling with a 2×2 filter and stride $= 2$ 13
2.4 Step function 14
2.5 Sigmoid function 15
2.6 Tanh function 15
2.7 ReLu function 16
2.8 LeakyReLu function 16
2.9 Illustration showing how a change in the weight $w_{i,j}$ of the neuron in the hidden layer $l-1$ influence the weight $w_{j,k}$ of the neron in the following layer l. 19
2.10 Illustration of gradient descent algorithm on a serise of level sets. Image is taken from Wikipedia. 25
2.11 SGD fluctuation. Figure is taken from Wikipedia 26
2.12 Trainging process between SGD (left), and SGD with momentum (right). 27
2.13 Generative Adversarial Network framework. 29
2.14 Adapted from the DCGAN paper [8]. The Generator network implemented here. Note the non-existence of fully connected and pooling layers. 31

3.1 Traversals along the smile vector. Train on CelebA dataset [9]. Quote from original paper [10]. 34
3.2 Demonstration on changing independent addtional dimension. When changing the addtional dimension on latent space, the generated sample will change accordingly. Quote from original paper [11]. 35

3.3 Basic structure of combing GAN with VAE. 36
3.4 The structure of BiGAN. Quote from original paper [12] 36
3.5 The convolutional filter learned by E, G, D, in BiGAN. Quote from original paper [12] 36
3.6 Illustration of the IVE-GAN architecture. Quote from original paper [13] 37
3.7 Visualization of the 2-dimensional t-SNE [14] of the latent space representation. Quote from original paper [13] 38
3.8 Idea of AGE. Green region: modelled by G; Blue region: modelled by E; Gray region: latent space; Objective: blue overlap with green, and be full of gray space. Quote from original paper [15] 38
3.9 The structure of SelfExGAN. Our model contains of three parts which are E, G, and D. $p_E(x, z)$ denotes the joint distribution learned by E, $p_G(z, x)$ denotes the joint distribution learned by G. $p_g(x)$ denotes the distribution learned by G for mapping z into x, $p_e(z)$ denotes the distribution learned by E for mapping x to z. $D1$ trying to distinguish the generated $\hat{x}$ and real x, $D2$ trying to distinguish the fake pair $(z, \hat{x})$, $(x, \hat{z})$, and real pair (z, x). 39

4.1 Results on MNIST dataset. 48
4.2 Results on SVHN dataset. 50
4.3 Results on Cifar10 dataset. 52
4.4 Results on Imagenet dataset. 55
4.5 Results on CelebA crop dataset. 57
4.6 Results on CelebA dataset. 59
4.7 Results on cartoon dataset. 61
4.8 Given a real image, then iteratively keep feeding with the former generated sample. 63
4.9 Visualization of the 2-dimensional t-SNE [14] of the latent space representation on MNIST dataset. 64
4.10 Generate data for supervised learning. 65
4.11 Similarity evaluation with human faces. 66
4.12 Improving the divergent Creation. 66

List of Tables

3.1 Classification accuracy of BiGAN for Imagenet dataset[16] compare with other methods. 37

4.1 MNIST model parameter setting. 49
4.2 SVHN model parameter setting. 51
4.3 Cifar10 model parameter setting. 53
4.4 Imagenet model parameter setting. 56
4.5 CelebA(with center crop) model parameter setting. 58
4.6 CelebA(with resized to 64×64) model parameter setting. 60
4.7 Cartoon model parameter setting. 62

Chapter 1

Introduction

In this chapter, we will explain some basic knowledge about Deep Neural Network (DNNs). We start by the basic concepts behind DNNs. Then we introduce several famous and important architectures which are LeNet [2], AlexNet [3], DeConvNet [4], VGGNet [5], GoogleNet [6], and ResNet [7], in chronological order.

1.1 Basic Concepts

The human brain neural network is a very complex nervous system. An artificial neural network (ANN) is a computational model based on the structure of biological neural networks. It mimics the way information is processed in the human brain. Typically, ANNs have an input layer, hidden layers and an output layer. In each layer, there are a set of connected units called artificial neurons.

It is estimated, that the average human brain contains 86 billion neurons [17]. Together they form a huge network. Even if we knew the detailed inner structure of the human brain, we would still not be able to simulate it with current technology because of its robustness. Our efforts are therefore rather different. We want to build a neural network with a good ratio between its size and its effectiveness.

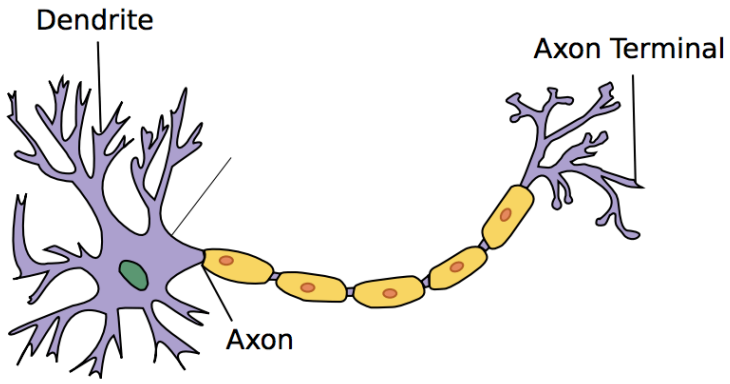


Fig. 1.1: Model of biological neuron. Image is taken from Wikipedia.

Deep Neural Network (DNN) refers to artificial neural networks that are composed of many layers. The earliest deep learning architecture is composed of multiple layers of non-linear features and it has polynomial activation functions. It is proposed by Ivaknenko and Lapa in 1965 [18]. In 1979, Fukushima [19] proposed the Convolutional Neural Networks (CNN), which has multiple pooling and convolutional layers. CNN has the advantage of avoiding the hand-selected feature extraction, which is time-consuming and requires expert knowledge, compared to the traditional image processing algorithms.

Genrally, DNNs consists of a set of artificial neurons. Formally, an aritificial neural has n inputs represented as a vector $\vec{x} \in \mathbb{R}^n$. Inputs in an artificial neuron correspond to the dendrites in a biological neuron, while a single output of an artificial neuron corresponds to the axon in a biological neuron, which is depicted in Figure 1.1. Each input $i, 1 \leq i \leq n$, has an assigned weight $w_1, w_2, ..., w_n$, and bias $b_1, b_2, ..., b_n$. Weighted input values are combined and followed by a non-linar activation (see in Chapter 2), as shown in Figure 1.2.

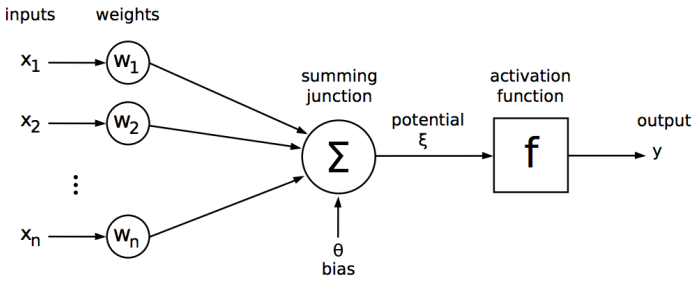


Fig. 1.2: Model of an artificial neuron

DNNs represent images using a multi-layered hierarchy of features and are inspired by the structure and functionality of the visual pathway of the human brain. Figure 1.3 [20] shows human's visual system. Visual information is collected by our eyes, and project to the retina. Then the information will be transmitted in a stream way (layers by layers. Actually, different layers in our brain servers as different functionality, in Brain Sciences, we divide it into V1, V2, V3, V4 regions [1].

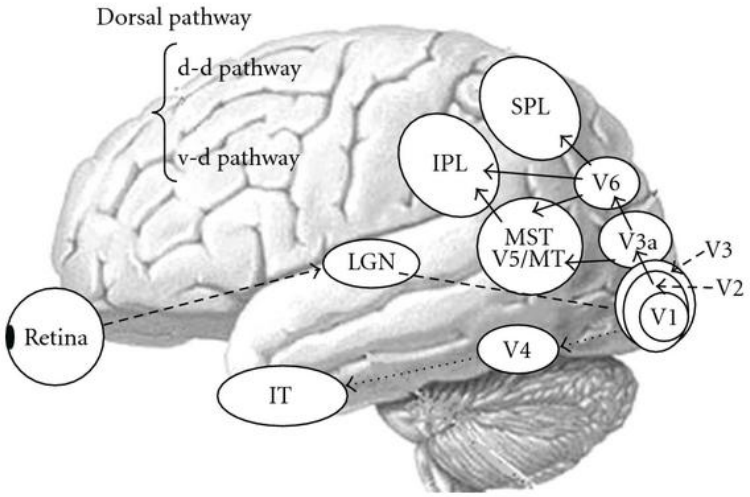


Fig. 1.3: Brain Structure [1]. Explain how human's visual system works.

DNNs also mimic the information flow in a layers by layers way. It works by feeding the data into the input neurons. The data flow in the direction of oriented edges and ends when the output neurons are hit. The result is interpreted from the values obtained in the output neurons. For the input neurons, they together represent an instance of the problem to be solved by the DNNs. All output neurons have exactly one output, and all outputs together represent a possible solution to the problem. Between input neurons and

output neurons, there exist a layers by layers structure. Figure 1.4 shows a Multilayer perceptron [21] network consists of three layers. A set of hidden neurons consists of the neurons which are not input, nor output neurons. Their number and organization into layers may vary even for the same problem, but is a key feature of the network vastly influencing its performance. The Multilayer perceptron (MLP) is a feed-forward neural network consisting of multiple mutually interconnected layers of neurons. The layers are stacked one onto each other. Every neuron in one layer is connected to every neuron in the following layer. The motivation behind designing multilayer networks is to be able to solve more complex tasks. The layers in MLP network is built of fully-connected layers defined in Chapter 2.

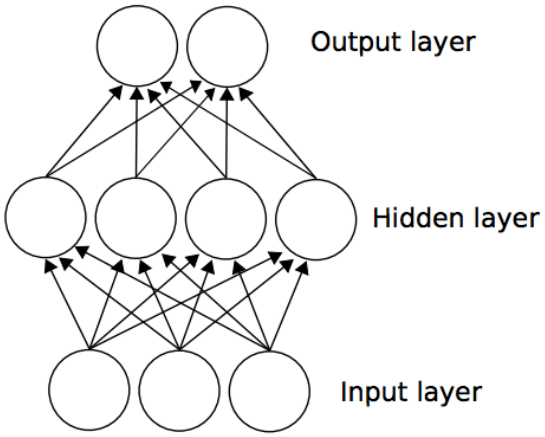


Fig. 1.4: A Multilayer perceptron network consists of three layers

Inspired by biological nervous systems, DNNs aim at reaching their versatility through learning. ANNs are commonly employed in artificial intelligence, machine learning and pattern recognition. There has been substantial research into how the human brain's structure achieves such a high level of versatility. This research has provided some important insights, however the conclusions are far from completely explaining the complex functioning of the brain. Even though we have not been able to replicate the brain so far, the field of artificial intelligence offers very effective solutions to many problems by simulating the observations of biological research of various nervous systems.

DNNs use the stacked layers structure to solve various problems in real world. For example, image classification, speech recognition, language transformation and so on. Based on my understanding, DNNs can be thought as a huge black box for high-dimensional approximator. The implementation of DNNs can be easily described by linear transformation followed by non-linear activation. By stacking these procedures again and again, DNNs are competent for a variety of tasks.

1.2 Famous DNN Architectures

In this section, we will introduce several important modern DNNs architectures. LeNet [2] the first successful application of Convolutional neuron networks (CNNs) to digit recognition, developed by Yann LeCun in 1990. It consists of a sequence of Convolutional, Max Pooling layers followed by a Fully Connected layer. AlexNet [3] popularized CNNs in computer vision, did really well on the ImageNet ILSVRC [16] challenge in 2012, showing significant gains in performance. The network has similar architecture to LeNet but is deeper and bigger and features convolutional layers stacked on top of each other. DeCon-

vNet [4] demonstrates how to visualize convolutional layer's learning results. VGG16 [5] demonstrated the importance of depth as a critical component to good performance; it was a runner-up in ILSVRC 2014. The architecture consists of a stacked convolutional and max-pooling layers, with increasing depth and it uses a large number of parameters due to the final fully connected layers. GoogLeNet [6] proposed from Google won the ILSVRC 2014 challenge. The architecture consists of inception modules that dramatically reduce the number of parameters; it uses multi-scale 3×3, 5×5 convolutional filters including 1x1 convolutions for dimensionality reduction. ResNet [7] residual network was the winner of ILSVRC 2015; it introduces skip connections for easier training that enable very deep architectures and makes use of batch normalization. Each of those famous modern structures will be introduced in the following subsections.

1.2.1 LeNet

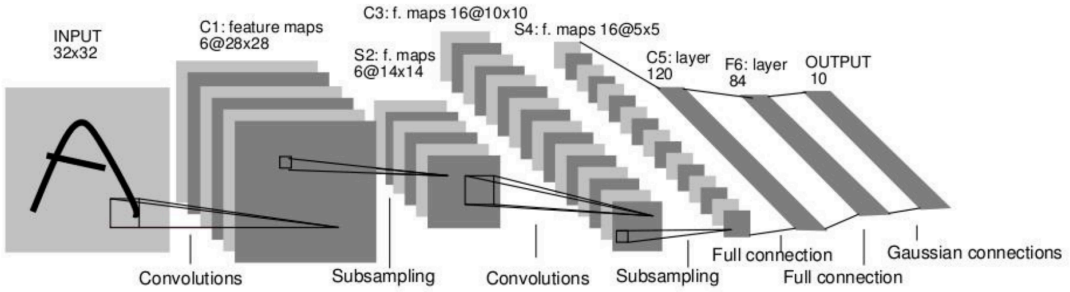


Fig. 1.5: Architecture of LeNet [2].

The LeNet architecture is an excellent "first architecture" for CNN (especially when trained on the MNIST dataset, an image dataset for handwritten digit recognition).

LeNet is small and easy to understand, yet large enough to provide interesting results. Furthermore, the combination of LeNet + MNIST is able to run on the CPU, making it easy for beginners to take their first step in Deep Learning and Convolutional Neural Networks. Figure 1.5 shows the architecture of LeNet. The LeNet architecture consists of the following layers which are convolutional layer, pooling layer, and fully-connected layer.

1.2.2 AlexNet

The one that started it all (Though some may say that Yann LeCun's paper in 1998 was the real pioneering publication). This paper, titled "ImageNet Classification with Deep Convolutional Networks" [3], has been cited a total of 17,955 times and is widely regarded as one of the most influential publications in the field. Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton created a "large, deep convolutional neural network" that was used to win the 2012 ILSVRC (ImageNet Large-Scale Visual Recognition Challenge). For those that aren't familiar, this competition can be thought of as the annual Olympics of computer vision, where teams from across the world compete to see who has the best computer vision model for tasks such as classification, localization, detection, and more. 2012 marked the first year where a CNN was used to achieve a top 5 test error rate of 15.4% (Top 5 error is the rate at which, given an image, the model does not output the

correct label with its top 5 predictions). The next best entry achieved an error of 26.2%, which was an astounding improvement that pretty much shocked the computer vision community. Safe to say, CNNs became household names in the competition from then on out.

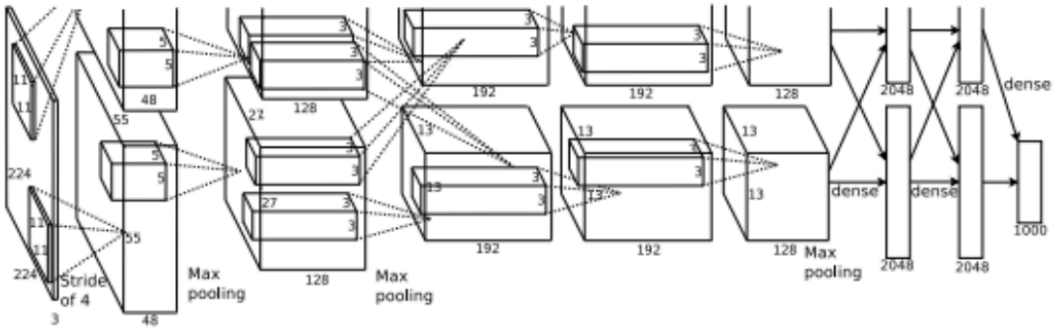


Fig. 1.6: Architecture of AlexNet [3].

In the paper, the group discussed the architecture of the network (which was called AlexNet). They used a relatively simple layout, compared to modern architectures. The network was made up of 5 conv layers, max-pooling layers, dropout layers, and 3 fully connected layers. The network they designed was used for classification with 1000 possible categories. Figure 1.6 shows the architecture of AlexNet. The Main points of AlexNet includes:

- Trained the network on ImageNet data, which contained over 15 million annotated images from a total of over 22,000 categories.
- Used data augmentation techniques that consisted of image translations, horizontal reflections, and patch extractions.
- Implemented dropout layers in order to combat the problem of overfitting to the training data.
- Trained the model using batch stochastic gradient descent, with specific values for momentum and weight decay.

The neural network developed by Krizhevsky, Sutskever, and Hinton in 2012 was the coming out party for CNNs in the computer vision community. This was the first time a model performed so well on a historically difficult ImageNet dataset. Utilizing techniques that are still used today, such as data augmentation and dropout, this paper really illustrated the benefits of CNNs and backed them up with record-breaking performance in the competition.

1.2.3 DeConvNet

The basic idea behind how this works is that at every layer of the trained CNN, you attach a "deconvnet" which has a path back to the image pixels. An input image is fed into the CNN and activations are computed at each level. This is the forward pass. Now, let's say we want to examine the activations of a certain feature in the 4th conv layer. We would store the activations of this one feature map, but set all of the other activations in the layer to 0, and then pass this feature map as the input into the deconvnet. This deconvnet has the same filters as the original CNN. This input then goes through a series of the un-pooling layer (reverse max-pooling layer), rectify, and filter operations for each

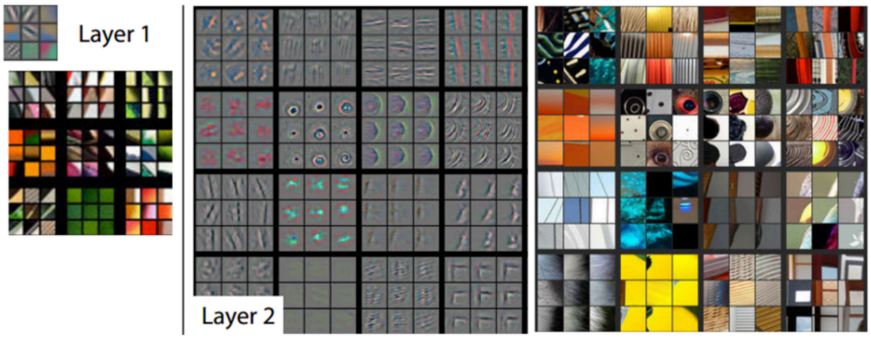


Fig. 1.7: Visualizations of layer 1 and layer 2 [4]. Each layer illustrated 2 pictures, one which shows the filters themselves and that shows what part of the image are most strongly activated by the given filter. For example, in the space labled layer 2, we have representation of the 16 different filters (on the left)

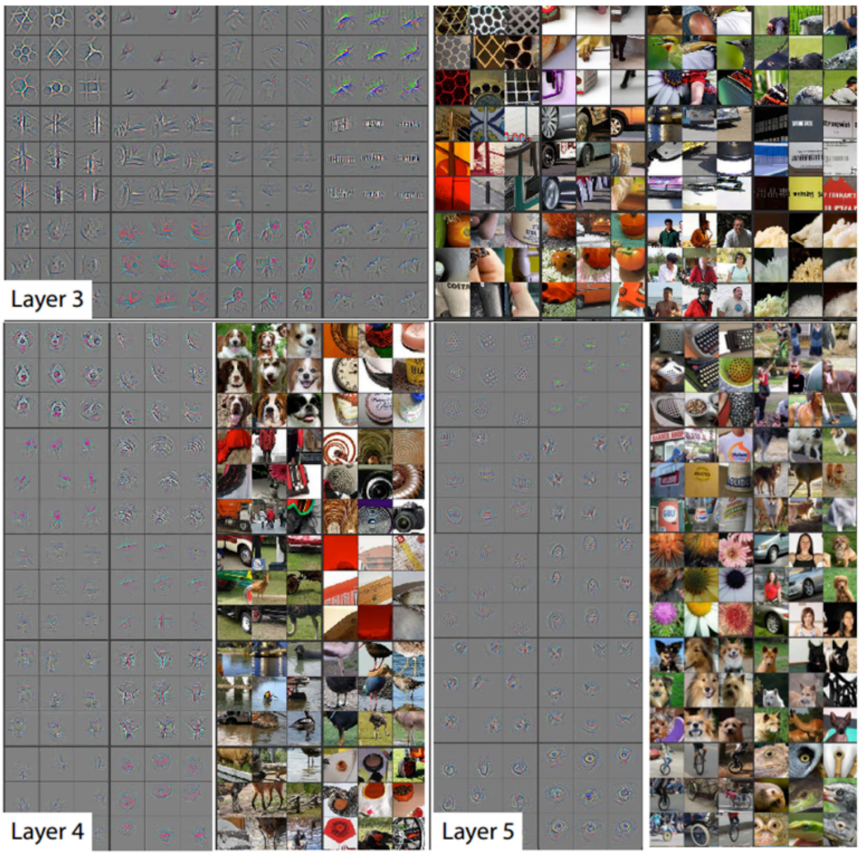


Fig. 1.8: More visualization on layer 3,4,5 [4].

preceding layer until the input space is reached. The reasoning behind this whole process is that we want to examine what type of structures excite a given feature map. Figure 1.7 shows the visualizations of the first and second layers.

The first layer of the ConvNet is always a low-level feature detector that will detect simple edges or colors in this particular case. Figure 1.8 gives the more visualization on 3,4,5 layers. These layers show a lot more of the higher level features such as dogs' faces or flowers. This means the proceeding layer try to learn more high-level features.

1.2.4 VGGNet

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Fig. 1.9: The 6 different architectures of VGGNet in original paper [5].

Figure 1.9 shows 6 different architectures of VGGNet. The keypoints in VGGNet includes:

- The use of only 3×3 sized filters is quite different from AlexNet's 11×11 filters in the first layer and DeConvNet's 7×7 filters. The authors' reasoning is that the combination of two 3×3 conv layers has an effective receptive field of 5×5. This in turn simulates a larger filter while keeping the benefits of smaller filter sizes. One of the benefits is a decrease in the number of parameters. Also, with two conv layers, we're able to use two ReLU layers instead of one.
- 3 conv layers back to back have an effective receptive field of 7×7.
- As the spatial size of the input volumes at each layer decrease (result of the conv and pool layers), the depth of the volumes increase due to the increased number of

filters as you go down the network.
- Interesting to notice that the number of filters doubles after each max-pooling layer. This reinforces the idea of shrinking spatial dimensions, but growing depth.

VGGNet is one of the most influential papers in my mind because it reinforced the notion that convolutional neural networks have to have a deep network of layers in order for this hierarchical representation of visual data to work. Keep it deep. Keep it simple.

1.2.5 GoogLeNet

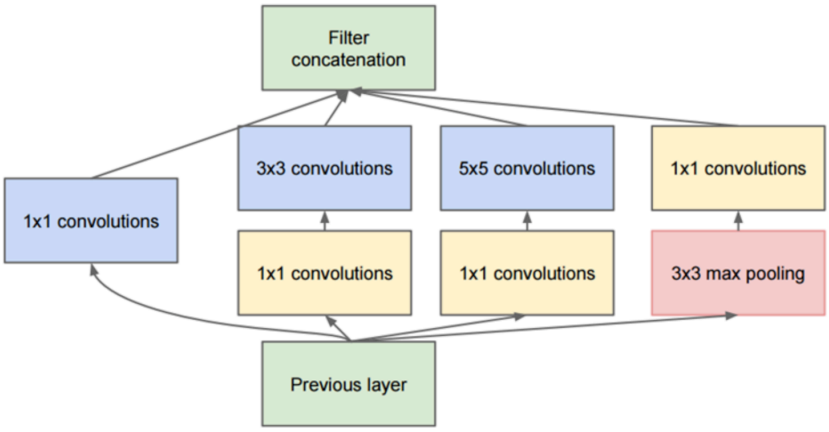


Fig. 1.10: Full inception module architecture [6].

The Inception module has been proposed by GoogLeNet [6]. Figure 1.10 shows the architecture of a full Inception. GoogLeNet is a 22 layer CNN and was the winner of ILSVRC 2014 with a top 5 error rate of 6.7%. To my knowledge, this was one of the first CNN architectures that really strayed from the general approach of simply stacking convolution and pooling layers on top of each other in a sequential structure. The authors of the paper emphasized that this new model places notable consideration on memory and power usage.

In Figure 1.10, the bottom green box is our input and the top one is the output of the model (Turning this picture right 90 degrees would let you visualize the model in relation to the last picture which shows the full network). Basically, at each layer of a traditional ConvNet, you have to make a choice of whether to have a pooling operation or a conv operation (there is also the choice of filter size). What an Inception module allows you to do is perform all of these operations in parallel. In an Inception module, we have a medium sized filter convolution, a largely sized filter convolution, and a pooling operation. The network in network convolution is able to extract information about the very fine grain details in the volume, while the 5×5 filter is able to cover a large receptive field of the input, and thus able to extract its information as well. It also has a pooling operation that helps to reduce spatial sizes and combat overfitting. On top of all of that, it has ReLus after each conv layer, which helps improve the nonlinearity of the network. Basically, the network is able to perform the functions of these different operations while still remaining computationally considerate.

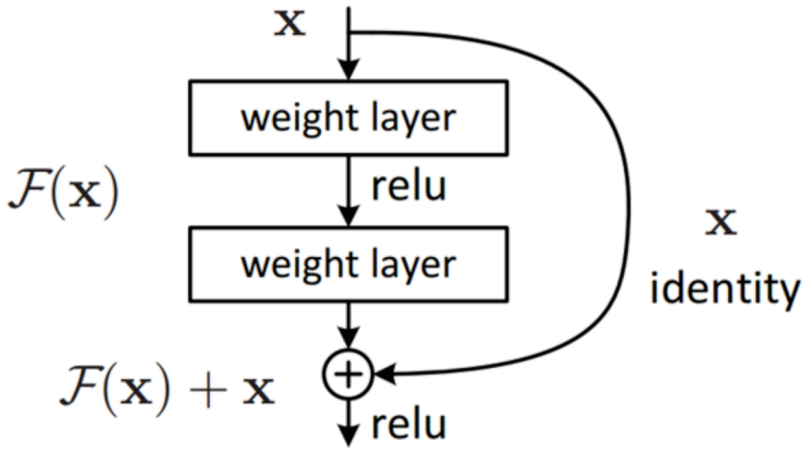


Fig. 1.11: A Residual Block [7].

1.2.6 ResNet

Imagine a deep CNN architecture. Take that, double the number of layers, add a couple more, and it still probably isn't as deep as the ResNet architecture that Microsoft Research Asia came up with in late 2015. ResNet is a new 152 layer (the deepest version) network architecture that set new records in classification, detection, and localization through one incredible architecture. Aside from the new record in terms of the number of layers, ResNet won ILSVRC 2015 with an incredible error rate of 3.6% (Depending on their skill and expertise, humans generally hover around a 5-10% error rate.

Except its depth, ResNet introduce a new module called Residual Block [7] see in Figure 1.11. The idea behind a Residual Block is that you have your input x go through conv-relu-conv series. This will give you some $F(x)$. That result is then added to the original input x. Let's call that $H(x) = F(x) + x$. In traditional CNNs, your $H(x)$ would just be equal to $F(x)$. So, instead of just computing that transformation (straight from x to $F(x)$), we're computing the term that you have to add, $F(x)$, to your input x. Basically, the mini-module shown below is computing a "delta" or a slight change to the original input x to get a slightly altered representation (When we think of traditional CNNs, we go from x to $F(x)$ which is a completely new representation that doesn't keep any information about the original x). The authors believe that it is easier to optimize the residual mapping than to optimize the original, unreferenced mapping [7].

1.3 Thesis Structure

The structure of this paper listed as follows:

In Chapter 2, the basic concepts of Generative Adversarial Network, which includes neural network architectures, layers, non-linear activation functions, loss functions and training methods are introduced.

In Chapter 3 , we introduce the proposed model: a latent space understandable Generative Adversarial Network. The general idea, structure and the theoretical results are presented in this chapter.

In Chapter 4, we exhibit the learning ability of our model on various dataset. Implementation detail and its applications wil be introduced as well.

In Chapter 5, the conclusions, and future works are presented.

Chapter 2

Generative Adversarial Networks

DNNs have a recursively layer by layer architecture. In this chapter, we introduce the components in DNNs, and how to train a DNN by seeking a set of parameters for the network that minimize the objective function. Also, Generative Adversarial Networks [22] (GAN) will be introduced in this chapter. GAN consists of two DNNs, which are called generator and discriminator separately. The main idea behind a GAN is to have these two competing neural network models trying to achieve opposite objectives. Generator takes random noise as input and tries to generate samples with high authenticity. On the other hand, discriminator receives samples from both the generator and the training data, and has to be able to distinguish between these two sources. G and D play a continuous game, where the generator is learning to produce more realistic samples, and the discriminator is learning to distinguish generated data from real one. These two networks are trained simultaneously, and the objective is that the competition will drive the generated samples to be indistinguishable from real ones. In Section 2.4, we introduce the architecture of GANs.

2.1 Layers

Even though there are many different architectures for DNNs in the literature, the majority of them can be built by stacking four main type of layers in different combinations. Namely, the fully connected layer, the convolutional layer, pooling layer and batch normalization layer. In this section, we will explain those layers.

2.1.1 Fully Connected Layer

Mathematically, we can think of a linear layer as a function which applies a linear transformation on a vectorial input of dimension I and output a vector of dimension O. Usually, the layer has a bias parameter:

$$y = W \cdot x + b \tag{2.1}$$

The linear layer is motivated by the basic computational unit of the brain called neuron. Approximately 86 billion neurons can be found in the human nervous system and they are connected with approximately $10^{14} - 10^{15}$ synapses [17]. Each neuron receives input signals from its dendrites and produces output signal along its axon. The linear layer is a simplification of a group of neurons having their dendrites connected to the same inputs. Usually, an activation function, such as sigmoid, is used to mimic the $1-0$ impulse carried away from the cell body and also to add non-linearity (See Figure 2.1).

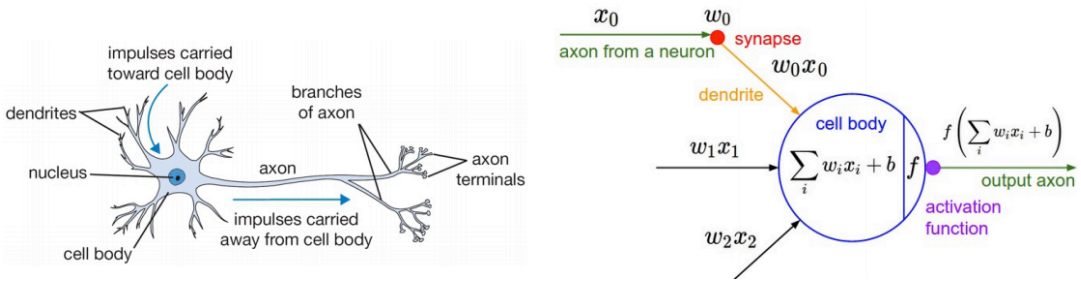


Fig. 2.1: A cartoon drawing of a biological neuron (left), and its mathematical model (right). Image is taken from Wikipedia.

2.1.2 Convolutional Layer

The purpose of convolutional layers is to detect the features in the presented images. It consists of multiple feature maps, each recognizing certain specific feature. The feature recognition can be thought of as running the sub-image through a filter. The filtering is essentially done through weight adjustments.

Regular neural networks, only made of linear and activation layers, do not scale well to full images. For instance, images of size $3 \times 224 \times 224$ (3 color channels, 224 width, 224 hight) would necessitate a first linear layer having $3 * 224 * 224 + 1 = 150129$ parameters for a single neurone (*e.g.* output). However, convolution layers take advantage of the fact that their input (*e.g.* images or feature maps) exhibits many spatial relationships. In fact, neighboring pixels should not be affected by their location within image. Thus, a convolutional layer learns a set of N_k filters $F = f_1, f_2, ..., f_{N_k}$, which are convolved spatially with input image x, to produce a set of N_k 2D features maps z:

$$z_k = f_k \otimes x \tag{2.2}$$

where $\otimes$ is the convolution operator. When the filter correlates well with a region of the input image, the response in the corresponding feature map location is strong. Unlike conventional linear layer, weights are shared over the entire image reducing the number of parameters per response and equivariance is learned (*i.e.* an object shifted in the input image will simply shift the corresponding responses in a similar way). Figure 2.2 shows 2×2 convolution.

Also, a fully connected layer can be seen as a convolutional layer with filter of sizes $1 \times 1 \times$ *input size*. In addition, in CNNs, each filter f_i is replicated across the entire visual field. These replicated units share the same parameterization (weight vector and bias) and form a feature map.

2.1.3 Deconvutional Layer

The concept of deconvolution (also be called transposed convolution) is first shown in the Zeiler's paper [4] published in 2010. The need for deconvolutions generally arises from the desire to use a transformation going in the opposite direction of a normal convolution. Deconvolutions work by swapping the forward and backward passes of a convolution.

Imagine inputting an image into a single convolutional layer. Now take the output, throw it into a black box and out comes your original image again. This black box does a deconvolution. It is the mathematical inverse of what a convolutional layer does.

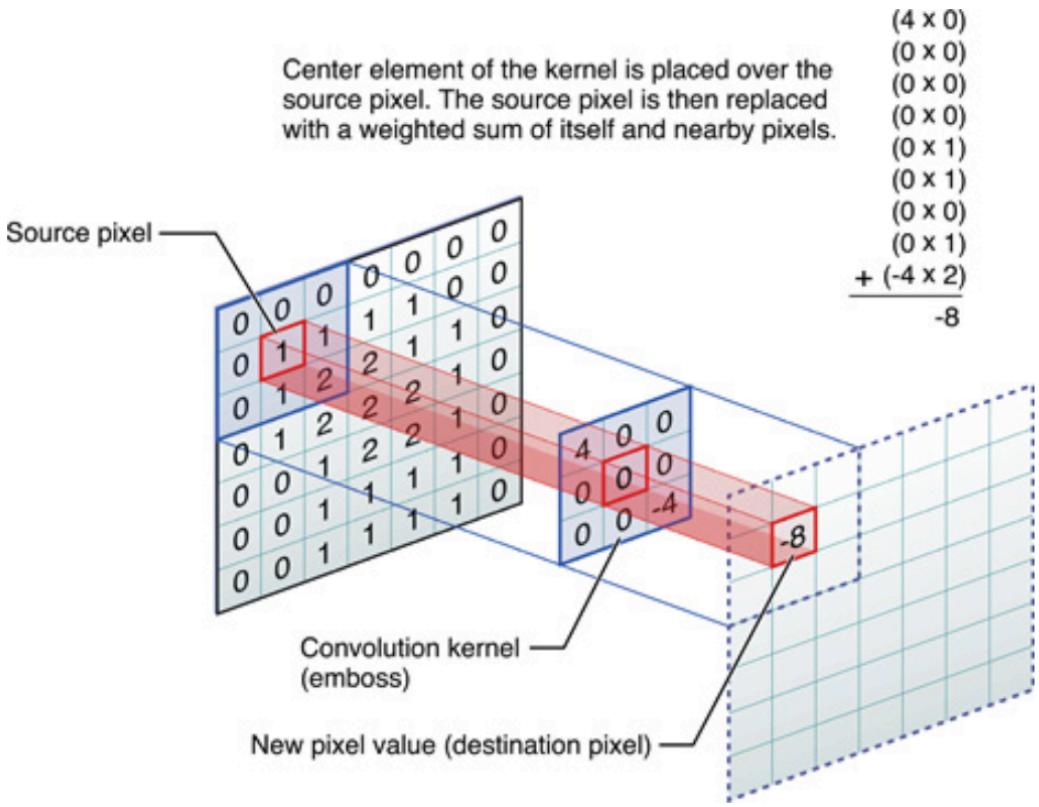


Fig. 2.2: Convolution with 2×2 filter. Image is taken from Wikipedia.

A deconvolution is somewhat similar because it produces the same spatial resolution a hypothetical deconvolutional layer would. However, the actual mathematical operation that's being performed on the values is different. A deconvolutional layer carries out a regular convolution but reverts its spatial transformation. Deconvolutional layers are adopted by many works, such as scene segmentation [23].

2.1.4 Pooling Layer

In CNNs, a pooling layer is typically present to provide invariance to slightly different input images and to reduce the dimension of the feature maps (*e.g.* width, hight):

$$P_R = P_{i \in R}(z_i) \tag{2.3}$$

where P is a pooling function over the region of pixels R.

Another important concept of CNNs is pooling, which is a form of non-linear down-sampling. There are several non-linear functions to implement pooling among which max pooling is the most common. Max pooling is preferred as it avoids cancellation of negative elements and prevents blurring of the activations and gradients throughout the network since the gradient is placed in a single location during backpropagation. Figure 2.3 shows an example of max pooling. In addition to max pooling, the pooling units can use other functions, such as average pooling.

The pooling layer serves to progressively reduce the spatial size of the representation, to reduce the number of parameters and amount of computation in the network, and hence to also control overfitting. It is common to periodically insert a pooling layer between

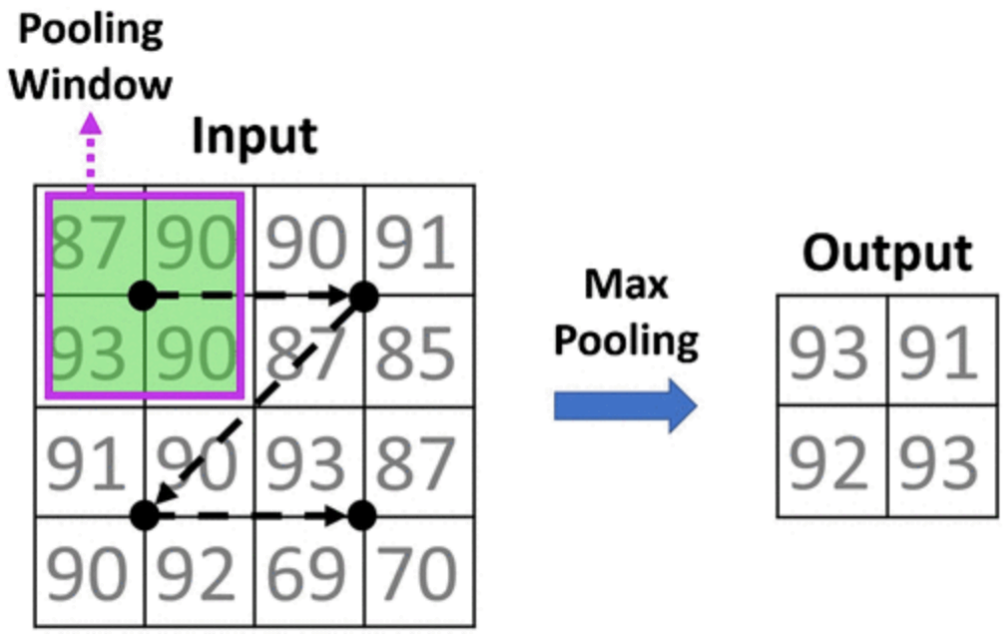


Fig. 2.3: Max pooling with a 2×2 filter and stride $= 2$

successive convolutional layers in a CNN architecture. The spatial pooling layer is defined by its aggregation function, the high and width dimensions of the area where it is applied, and the properties of the convolution (e.g. padding, stride).

2.1.5 Batch Normalization Layer

The Batch Normalization (BN) layer [24] quickly became very popular mostly because it helps to converge faster. It adds a normalization step (shifting inputs to zero-mean and unit variance) to make the inputs of each trainable layers comparable across features. By doing this, it ensures a high learning rate while keeping the network learning.

2.2 Activation Functions

The capacity of the neural networks to approximate any functions, especially non-convex, is directly the result of the non-linear activation functions. Every kind of activation function takes a vector and performs a certain fixed point-wise operation on it. Mainly, all the activation functions can be divided into two groups, which are free-parametric activation function and parametric activation functions. In this section, we will introduce various kinds of non-linear activation function, and how they work.

2.2.1 Non-parametric Activation Functions

In this section, we will introduce some basic activation function with no parameters need to learn which are Step function, Sigmoid function, Tanh function, ReLu function, LeakyReLu function [25].

Step Function

$$y = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases} \tag{2.4}$$

Fig. 2.4: Step function

Sigmoid Function

$$y = sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{2.5}$$

Tanh Function

$$y = tanh(x) = \frac{2}{1 + e^{-2x}} - 1 = 2sigmoid(2x) - 1 \tag{2.6}$$

ReLu Function

$$y = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases} \tag{2.7}$$

LeakyReLu Function

$$y = \begin{cases} x & x > 0 \\ \lambda x & x \leq 0 \end{cases} \tag{2.8}$$

where λ is a small value, often set to 0.2.

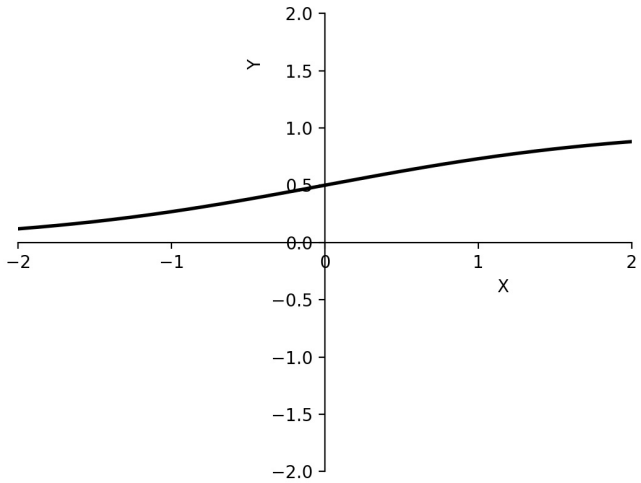


Fig. 2.5: Sigmoid function

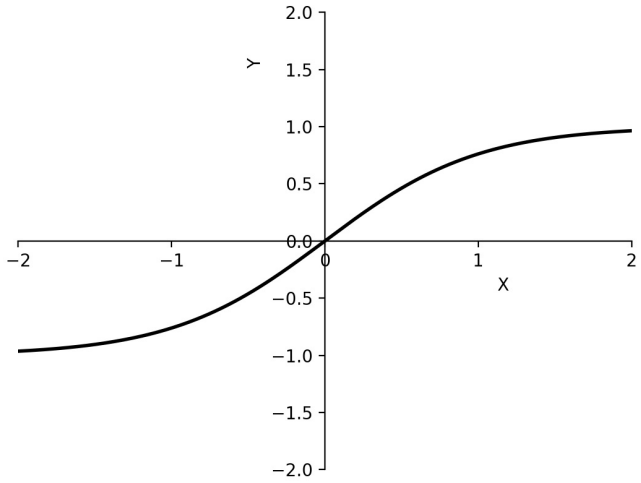


Fig. 2.6: Tanh function

2.2.2 Parameterized Activation Functions

In this section, we will introduce another type of non-linear activation function: parametric activation function. PReLu [26] (Parametric Rectified Linear Unit).

PReLu

$$y_i = \begin{cases} x_i & x_i > 0 \\ \lambda_i x & x_i \leq 0 \end{cases} \tag{2.9}$$

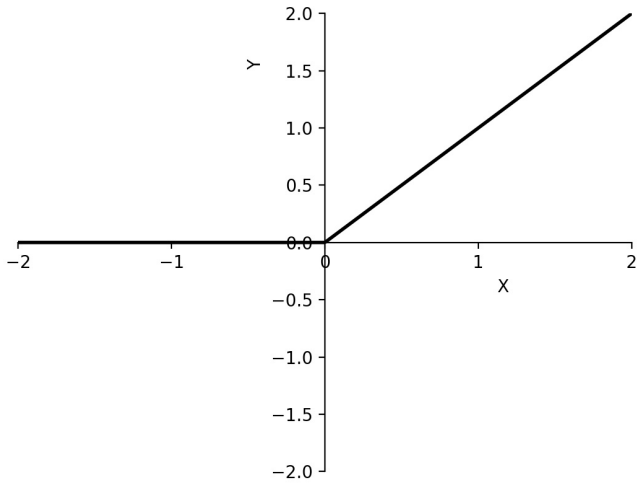


Fig. 2.7: ReLu function

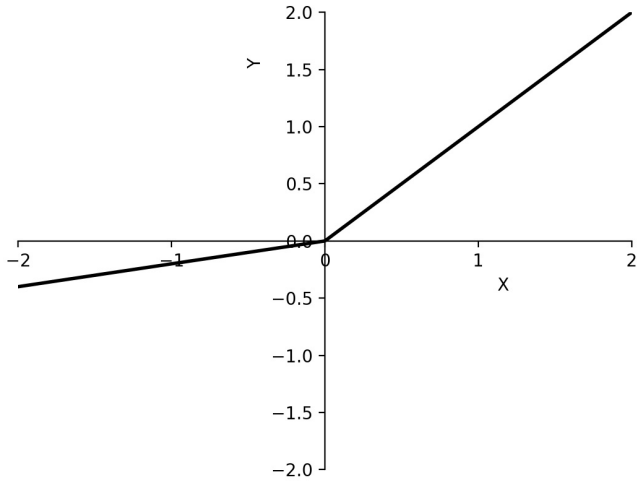


Fig. 2.8: LeakyReLu function

As its name suggests, λ_i are parameter need to be learn. If we set λ_i to 0, than PReLu becomes ReLu. If we set λ_i to a small fixed value, it becomes LeakyReLu.

2.3 Training

The ability to learn is the key concept of neural networks. The aim of the process is to find the optimal parameters (and structure) of the network for solving the given task. Before the training process starts, network parameters need to be initialized. Initial values are often chosen randomly, however using some heuristics may lead to a faster parameter adjustment towards the optimal values. Learning is then carried out on the training set

by feeding the training data through the network. It is an iterative process, where the outputs produced on each input from the training set are analyzed and the network is repeatedly being adjusted to produce better results. The network is considered to be trained after reaching the target performance on the training data.

The networks will be trained based on gradient back propagation [27]. Namely, by computing the difference between the output of a neuron network and the expected output, we keep propagating the error back through the whole network guiding the parameters updating. Section 2.3.1 will give more details on back propagation algorithm. How to define "difference" based on the loss function we define for the network, which will be introduced in Section 2.3.2. When we updating the network parameters, we can choose various updating strategy to update the network parameters. In Section 2.3.3, we will introduce some most often used optimization algorithm for updating parameters.

2.3.1 Back Propagation

Back propagation is a method used in artificial neural networks to calculate the error contribution of each neuron after a batch of data (in image recognition, multiple images) is processed. It is a special case of an older and more general technique called automatic differentiation. In the context of learning, back propagation is commonly used by the gradient descent optimization algorithm to adjust the weight of neurons by calculating the gradient of the loss function. This technique is also sometimes called backward propagation of errors, because the error is calculated at the output and distributed back through the network layers.

The back propagation algorithm has repeatedly been rediscovered and is equivalent to automatic differentiation in reverse accumulation mode. Back propagation requires a known, desired output for each input value. It is therefore considered to be a supervised learning method. Back propagation is also a generalization of the delta rule to multilayered feedforward networks, made possible by using the chain rule to iteratively compute gradients for each layer. It is closely related to the GaussNewton algorithm, and is part of continuing research in neural backpropagation.

To define the learning process formally and in more details in following sections. Let us consider P training patterns labeled $(\vec{x^p}, \vec{d^p})$, where $\vec{x^p}$ is the desired output vector, and $1 \leq p \leq P$. Given the current configuration of the network, the input $\vec{x^p}$ yields the output $\vec{y^p}$. Then for every pattern p we want $\vec{y^p}$ to be as close to the desired output $\vec{d^p}$ as possible. Then, we are able to define the error of each neuron j in the output layer as:

$$e_j^p = y_j^p - d_j^p \tag{2.10}$$

Now we can define the squared error for pattern p as:

$$E_p = \frac{1}{m_\kappa} \sum_{j=1}^{m_\kappa} (e_j^p)^2 = \frac{1}{m_\kappa} (y_j^p - d_j^p)^2 \tag{2.11}$$

where m_κ is the number of neurons in the output layer. Note that if the actual output is exactly the same as the desired output, we get zero for the squared error. In other saying, the following holds true:

$$\forall j : E_p = 0 \rightleftharpoons e_j^p = 0 \rightleftharpoons y_j^p = d_j^p \tag{2.12}$$

It may be useful to sum up the average error for all input patterns to assess the network performance on the whole dataset, which can be achieved simply by computing the mean

squared error:

$$E_{avg} = \frac{1}{P}\sum_{p=1}^{P} E_p \tag{2.13}$$

When learning, for each interconnected pair of neurons (i, j), where i is a neuron in layer l, j is a neuron in layer $l+1$ and $w_{i,j}$ weights their connection, we want to adjust $w_{i,j}$ to minimize the mean squared error E_{avg}. Provided the activation function is differentiable everywhere on its domain, Eavg is also differentiable. When adjusting the weight $w_{i,j}$ of the neuron j located in the output layer κ, we are therefore interested in the partial derivative:

$$\frac{\partial E_{avg}}{\partial w_{i,j}} = \frac{1}{P}\frac{\partial}{\partial w_{i,j}}\sum_{p=1}^{P} E_p = \frac{1}{P}\sum_{p=1}^{P}\frac{\partial E_p}{\partial w_{i,j}} \tag{2.14}$$

To be able to adjust the network after each presented input pattern, we are actually interested in the derivative for each given pattern p and its corresponding E_p. In the following equations we will therefore omit the pattern index p. Applying the chain rule to Equation (2.14) we can get:

$$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial y_j}\frac{y_j}{w_{i,j}} \tag{2.15}$$

Using Equation (2.11) we can evaluate the first factor as:

$$\frac{E}{\partial y_j} = (y_j - d_j) \tag{2.16}$$

Then, we can evaluate the second factor:

$$\frac{\partial y_j}{\partial w_{i,j}} = \frac{\partial y_j}{\partial \xi_j}\frac{\partial \xi_j}{\partial w_{i,j}} = f^{'}(\xi_j)\frac{\partial}{\partial w_{i,j}}\sum_{k} w_{k,j}y_k = f^{'}(\xi_j)y_i \tag{2.17}$$

By combining both evaluated factors we can get:

$$\frac{\partial E}{\partial w_{i,j}} = (y_j - d_j)f^{'}(\xi_j)y_i \tag{2.18}$$

For convenience local gradient term δ_j for neuron j in the output layer as the following relation:

$$\delta_j = \frac{\partial E}{\partial \xi_j} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial \xi_j} = (y_j - d_j)f^{'}(\xi_j) \tag{2.19}$$

Then from this, we can convert Equation (2.18) into the following:

$$\frac{\partial E}{\partial w_{i,j}} = \delta_j y_i \tag{2.20}$$

By applying Equation (2.20) we can compute the gradient of the error function for each of the given patterns p. We need to adjust the weight $w_{i,j}$ proportionally to the gradient but in the opposite direction. However doing so for every input pattern would produce a

very unstable system. To combat this problem we can use a learning parameter $0 < \eta < 1$. The weight adjustment is then computed by:

$$\Delta w_{i,j} = -\eta \frac{\partial E}{\partial w_{i,j}} = -\eta \delta_j y_i \tag{2.21}$$

The weight adjustment $\Delta w_{i,j}$ in Equation (2.21) is only applicable to the neurons in the output layer. Computation of the adjustment for neurons in the hidden layers is more complicated. For instance, consider 3 neurons i, j, and k, all following each other on the same path along the layers $l-1$, l and $l+1$, respectively, as illustrated in Figure 2.9. Then the adjustment of $w_{i,j}$ needs to be done carefully, because besides influencing the output of neuron i itself, it also impacts all the outputs (and thus errors) in all layers following l. Minding this, let us bring our attention to the layers $l < L$ in the following text.

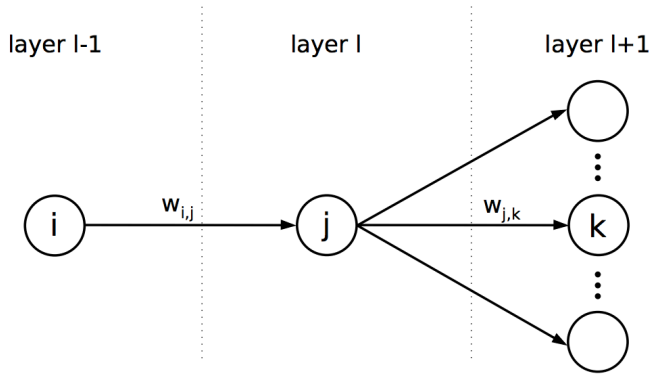


Fig. 2.9: Illustration showing how a change in the weight $w_{i,j}$ of the neuron in the hidden layer $l-1$ influence the weight $w_{j,k}$ of the neron in the following layer l.

Note that Equation (2.18) still applies to hidden layers. However, we need to look at the definition of the local gradient again. In the previous Equation (2.19), we are using the desired output d_j to calculate $\partial E/\partial y_j$. Of course, there is no desired output known in hidden layers. It is actually dependent on the network design. Because of this, we need to step back and use the following definition for δ_j:

$$\delta_j = \frac{\partial E}{\partial \xi_j} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial \xi_j} = \frac{\partial E}{\partial y_j} f'(\xi_j) \tag{2.22}$$

Now we need to redefine $\partial E/\partial y_j$ for hidden layers. For any hidden layer l, the following layer $l+1$ must exist (otherwise l would be the output layer). Given the neurons i, j and k each in a different layer as illustrated in Figure 2.9, we can use the potential ξ_k:

$$\frac{\partial E}{\partial y_j} = \sum_{k=1}^{m_{l+1}} \frac{\partial E}{\partial \xi_k}\frac{\partial \xi_k}{\partial y_j} = \sum_{k=1}^{m_{l+1}} \frac{\partial E}{\partial \xi_k} w_{j,k} = \sum_{k=1}^{m_{l+1}} \delta_k w_{j,k} \tag{2.23}$$

combining Equation (2.22) and Equation (2.23), we can get:

$$\delta_j = (\sum_{k=1}^{m_{l+1}} \delta_k w_{j,k}) f'(\xi_j) \tag{2.24}$$

From Equation (2.24), we can know that the local gradient of neuron k in layer $l+1$, we can calculate the local gradient for neuron j in layer l. This fact will allow us to recursively adjust the network weights going layer by layer in the direction from the output to the input (backwards).

Finally, we can summarize the parameter adjustment applicable to all the layers in a given network:

$$\Delta w_{i,j} = -\eta \frac{\partial E}{\partial w_{i,j}} = -\eta \delta_j y_i \tag{2.25}$$

where

$$\begin{aligned} \forall j \ in \ layer \ l < L \quad \delta_j &= (\sum_{k=1}^{m_{l+1}} \delta_k w_{j,k}) f^{'}(\xi_j) \\ \forall j \ in \ layer \ L \quad \delta_j &= (y_j - d_j) f^{'}(\xi_j) \end{aligned} \tag{2.26}$$

The above explanation is based on a fully-connected layer with no bias added. More details about derivation can be found on [28]. It is the same case in the other types of layers. The idea is that we compute the error between the output of network and our expected value. Then, we back propagate the gradient from behind to the head of the neuron network for updating the weights.

2.3.2 Loss Functions

Loss function is an important part in training DNNs, which is used to measure the inconsistency between predicted value $\hat{y}$ and actual label y. It is a non-negative value, where the robustness of model increases along with the decrease of the value of loss function. Loss function is the hard core of empirical risk function as well as a significant component of structural risk function. Generally, the structural risk function of a model is consist of empirical risk term and a regularization term, which can be represented as:

$$\theta^* = \arg\min_{\theta} \pounds(\theta) + \lambda \cdot \Phi(\theta) = \tag{2.27}$$

$$\arg\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} L(y^i, \hat{y^i}) + \lambda \cdot \Phi(\theta) = \tag{2.28}$$

$$\arg\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} L(y^i, f(x^i, \theta)) + \lambda \cdot \Phi(\theta) \tag{2.29}$$

Where $\Phi(\theta)$ is the regularization term or penalty term, θ is the parameters of model to be learned, $f(.)$ represents the activation function and $x^i = \{x_1^i, x_2^i, ..., x_m^i\}$ denotes the a training sample.

In this context, we only concentrate on the empirical risk term (loss function):

$$\pounds(\theta) = \frac{1}{n} \sum_{i=1}^{n} L(y_i, f(x^i, \theta)) \tag{2.30}$$

Mean Squared Error

Mean Squared Error (MSE), or quadratic, loss function is widely used in linear regression as the performance measure, and the method of minimizing MSE is called Ordinary

Least Squares (OSL), the basic principle of OSL is that the optimized fitting line should be a line which minimizes the sum of distance of each point to the regression line, *i.e.*, minimizes the quadratic sum. The standard form of MSE loss function is defined as:

$$\pounds = \frac{1}{n}\sum_{i=1}^{n}(y^i - \hat{y^i})^2 \tag{2.31}$$

where $(y^i - \hat{y^i})$ is named as residual, and the target of MSE loss function is to minimize the residual sum of squares. It is worthing to mention that if using sigmoid function as the non-linear activation function, the quadratic loss function would suffer the problem of slow convergence (learning speed), for other activation functions, it would not have such problem.

For example, by using sigmoid:

$$\hat{y^i} = \sigma(z^i) = \sigma(\theta^T x^i) \tag{2.32}$$

Simply case, if we only consider one sample, say that, $(y - \sigma(z))^2$, and it derivative can be computed by the following:

$$\frac{\partial \pounds}{\partial \theta} = -(y - \sigma(z)) \cdot \sigma^{'}(z) \cdot x \tag{2.33}$$

according to the shape and feature of Sigmoid, when $\sigma(z)$ tends to 0 or 1, $\sigma(z)^{'}$ is close to zero, and when $\sigma(z)$ close to 0.5, $\sigma(z)^{'}$ will reach it maximum. In this case, when the difference between predicted value and true label $(y - \sigma(z))$ is large, $\sigma(z)^{'}$ will close to 0, which decreases the convergence speed, this is improper, since we expect that the learning speed should be fast when the error is large.

Mean Squared Logarithmic Error

Based on our understanding, Mean Squared Logarithmic Error (MSLE) loss function is a variant of MSE, which is defined as:

$$\pounds = \frac{1}{n}\sum n(log(y^i + 1) - log(\hat{y^i} + 1))^2 \tag{2.34}$$

MSLE is also used to measure the difference between actual and predicted. By taking the *log* of the predictions and actual values, what changes is the variance that you are measuring. It is usually used when you do not want to penalize huge differences in the predicted and the actual values when both predicted and true values are huge numbers. Another thing is that MSLE penalizes under-estimates more than over-estimates.

L2 norm

L2 loss function is the square of the L2 norm of the difference between actual value and predicted value. It is mathematically similar to MSE, only do not have division by n, it is computed by the following:

$$\pounds = \sum_{i=1}^{n}(y^i - \hat{y^i})^2 \tag{2.35}$$

L2 is often used in various aspect, for more details, typocally in mathematic, please refer to [29] which gives more comprehensive explanation about L2 loss.

Mean Absolute Error

Mean Absolute Error (MAE) is a quantity used to measure how close forecasts or predictions are to the eventual outcomes, which is computed by the following:

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}|y^i - \hat{y^i}| \tag{2.36}$$

where $|\cdot|$ denotes the absolute value. Albeit, both MSE, and MAE are used in predictive modeling, there are several differences between them. MSE has nice mathematical properties which make it easier to compute the gradient. However, MAE requires more complicated tools such as linear programming to compute the gradient. Because of the square, large errors have a relatively greater influence on MSE than do the smaller error. Therefore, MAE is more robust to outliers since it does not make use of square. On the other hand, MSE is more useful if concerning about large errors whose consequences are much bigger than equivalent smaller ones. MSE also corresponds to maximizing the likelihood of Gaussian random variables.

L1 norm

L1 loss function is a sum of absolute errors of the difference between actual value and predicted value. Similar to the relation between MSE and L2, L1 is mathematically similar to MAE, only do not have division by n, and it is defined as the following:

$$\mathcal{L} = \sum_{i=1}^{n}|y^i - \hat{y^i}| \tag{2.37}$$

Kullback Leibler Divergence

Kullback Leibler Divergence (KL), also known as relative entropy, information divergence/gain, is a measure of how one probability distribution diverges from a second expected probability distribution. KL divergence loss function is computed by the following:

$$\mathcal{L} = \underbrace{\frac{1}{n}\sum_{i=1}^{n}(y^i \cdot log(y^i))}_{\text{entropy}} - \underbrace{\frac{1}{n}\sum_{i=1}^{n}(y^i \cdot log(\hat{y^i}))}_{\text{cross\_entropy}} \tag{2.38}$$

where the first term is entropy and another is cross entropy (another kind of loss function which will be introduced later). KL divergence is a distribution-wise asymmetric measure and thus does not qualify as a statistical metric of spread. In the simple case, a KL divergence of 0 indicates that we can expect similar, if not the same, behavior of two different distributions, while a KL divergence of 1 indicates that the two distributions behave in such a different manner that the expectation given the first distribution approaches zero.

Cross Entropy

Cross Entropy is commonly-used in binary classification (labels are assumed to take values 0 or 1) as a loss function (For multi-classification, use Multi-class Cross Entropy), which is computed by the following:

$$\mathcal{L} = -\frac{1}{n}\sum_{i=1}^{n}[y^i log(\hat{y^i}) + (1-y^i)log(1-\hat{y^i})] \tag{2.39}$$

Cross entropy measures the divergence between two probability distribution, if the cross entropy is large, which means that the difference between two distribution is large, while if the cross entropy is small, which means that two distribution is similar to each other. As we have mentioned in MSE that it suffers slow divergence when using sigmoid as activation function, here the cross entropy does not have such problem. Samely, $\hat{y^i} = \sigma(z^i) = \sigma(\theta^T x^i)$, and we only consider one training sample, by using sigmoid, we have $\pounds = ylog(\sigma(z)) + (1-y)log(1-\sigma(z))$, and compute it derivative as the following:

$$\frac{\partial \pounds}{\partial \theta} = (y - \sigma(z)) \cdot x \tag{2.40}$$

compare to the derivative in MSE, it eliminates the term $\sigma(z)^{'}$, where the learning speed is only controlled by $(y - \sigma(z))$. In this case, when the difference between predicted value and actual value is large, the learning speed, *i.e.*, convergence speed, is fast, otherwise, the difference is small, the learning speed is small, this is our expectation. Generally, comparing to a quadratic cost function, cross entropy cost function has the advantages that fast convergence and is more likely to reach the global optimization.

Negative Logarithmic Likelihood

Negative Log Likelihood loss function is widely used in neural networks, it measures the accuracy of a classifier. It is used when the model outputs a probability for each class, rather than just the most likely class. It is a "soft" measurement of accuracy that incorporates the idea of probabilistic confidence. It is intimately tied to information theory. And it is similar to cross entropy (in binary classification) or multi-class cross entropy (in multi-classification) mathematically. Negative log-likelihood is computed by the following:

$$\pounds = -\frac{1}{n}\sum_{i=1}^{n} log(\hat{y^i}) \tag{2.41}$$

Poisson

Poisson loss function is a measure of how the predicted distribution diverges from the expected distribution, the Poisson as loss function is a variant from Poisson distribution, where the Poisson distribution is widely used for modeling count data. It can be shown to be the limiting distribution for a normal approximation to a binomial where the number of trials goes to infinity and the probability goes to zero and both happen at such a rate that np is equal to some mean frequency for the process. The Poisson loss function is computed by the following:

$$\pounds = \frac{1}{n}\sum_{i=1}^{n} (\hat{y^i} - y^i \cdot log(\hat{y^i})) \tag{2.42}$$

Cosine Proximity

Cosine Proximity loss function computes the cosine proximity between predicted value and actual value, which is defined as:

$$\pounds = -\frac{y \cdot \hat{y}}{||y||_2 \cdot ||\hat{y}||} \tag{2.43}$$

where $\mathbf{y} = \{y^1, y^2, ..., y^n\} \in \mathbb{R}^n$, and $\hat{\mathbf{y}} = \{\hat{y}^1, \hat{y}^2, ..., \hat{y}^n\} \in \mathbb{R}^n$. It is same as Cosine Similarity, which is a measure of similarity between two non-zero vectors of an inner

product space that measures the cosine of the angle between them. In this case, note that unit vectors are maximally "similar" if they're parallel and maximally "dissimilar" if they're orthogonal (perpendicular). This is analogous to the cosine, which is unity (maximum value) when the segments subtend a zero angle and zero (uncorrelated) when the segments are perpendicular.

Hinge

Hinge Loss, also known as max-margin objective, is a loss function used for training classifiers. The hinge loss is used for "maximum-margin" classification, most notably for support vector machines (SVMs) [30]. For an intended output $y(i) = \pm\ 1$, *i.e.*, binary classification and a classifier score $\hat{y^i}$, the hinge loss of the prediction y is defined as

$$\pounds = \frac{1}{n}\sum_{i=1}^{n} max(0, 1 - y^i \cdot \hat{y^i}) \tag{2.44}$$

Note that $\hat{y^i}$ should be the "raw" output of the classifier's decision function, not the predicted class label. It can be seen that when y^i and $\hat{y^i}$ have the same sign (meaning $\hat{y^i}$ predicts the right class) and $|\hat{y^i}| > 1$, the hinge loss equals to zero, but when they have opposite sign, hinge loss increases linearly with $\hat{y^i}$ (one-sided error). However, there is a more general expression as the following:

$$\pounds = \frac{1}{n}\sum_{i=1}^{n} max(0, m - y^i \cdot \hat{y^i}) \tag{2.45}$$

where m margin is a customized value. More details about extending to multi-classification.

2.3.3 Optimization Algorithms

In this section, we will introduce some basic algorithms for finding the optimal weights when training DNNs. This section focuses on one particular case of optimization: nding the parameters of a neural network that signicantly reduce a cost function $J(\theta)$, which typically includes a performance measure evaluated on the entire training set. We will start with introducing the most fundamental algorithm Gradient Decent, then some more advanced algorithm will be introduced.

Gradient Descent

Gradient Descent (GD) is a first-order iterative optimization algorithm for finding the minimum of a function. To find a local minimum of a function using gradient descent, one takes steps proportional to the negative of the gradient (or of the approximate gradient) of the function at the current point. Gradient descent is one of the most popular algorithms to perform optimization and by far the most common way to optimize neural networks. At the same time, every state-of-the-art Deep Learning library contains implementations of various algorithms to optimize gradient descent (*e.g.* tensorflow's [31], theano' [32], and caffe's [33] documentation). Figure 2.10 illustrates the GD algorithm on a serise of level sets.

Gradient descent is based on the observation that if the multi-variable function $F(\mathbf{x})$ is defined and differentiable in a neighborhood of a point $\mathbf{a}$, then $F(\mathbf{x})$ decreases fastest if one goes from $\mathbf{a}$ in the direction of the negative gradient of F at $\mathbf{a}$, $-\nabla F(\mathbf{a})$). It follows that, if:

$$\mathbf{a}_{n+1} = \mathbf{a}_n - \gamma \nabla F(\mathbf{a}_n) \tag{2.46}$$

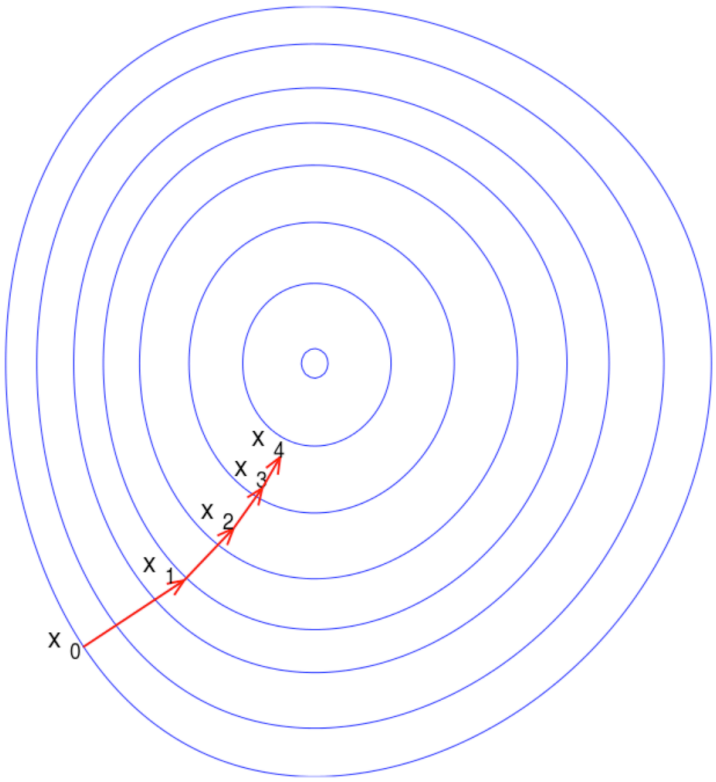


Fig. 2.10: Illustration of gradient descent algorithm on a serise of level sets. Image is taken from Wikipedia.

for γ small enough, then $F(\mathbf{a_n}) \geq F(\mathbf{a_{n+1}})$. In other words, the term $\gamma \nabla F(\mathbf{a})$ is subtracted from $\mathbf{a}$ because we want to move against the gradient, toward the minimum. With this observation in mind, one starts with a guess $\mathbf{x}_0$ for a local minimum of F, and considers the sequence $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots$, such that:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma_n \nabla F(\mathbf{x}_n), \ n \geq 0 \tag{2.47}$$

we have:

$$F(\mathbf{x}_0) \geq F(\mathbf{x}_1) \geq F(\mathbf{x}_2) \geq \cdots \tag{2.48}$$

So hopefully the sequence $(\mathbf{x}_n)$ converges to the desired local minimum. Note that the value of the step size γ is allowed to change at every iteration. With certain assumptions on the function F and particular choices of γ. The following term:

$$\gamma_n = \frac{(\mathbf{x}_n - \mathbf{x}_{n-1})^T [\nabla F(\mathbf{x}_n) - \nabla F(\mathbf{x}_{n-1})]}{||\nabla F(\mathbf{x}_n) - \nabla F(\mathbf{x}_{n-1})||^2} \tag{2.49}$$

convergence to a local minimum can be guaranteed. When the function F is convex, all local minima are also global minima, so in this case, gradient descent can converge to the global solution.

This process is illustrated in the adjacent picture (Figure 2.10). Here F is assumed to be defined on the plane, and that its graph has a bowl shape. The blue curves are the contour lines, that is, the regions on which the value of F is constant. A red arrow

originating at a point shows the direction of the negative gradient at that point. Note that the (negative) gradient at a point is orthogonal to the contour line going through that point. We see that gradient descent leads us to the bottom of the bowl, that is, to the point where the value of the function F is minimal.

Stochastic Gradient descent

Stochastic gradient descent (SGD) in contrast performs a parameter update for each training example $x(i)$ and label $y(i)$:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^i; y^i) \tag{2.50}$$

batch gradient descent performs redundant computations for large datasets, as it recomputes gradients for similar examples before each parameter update. SGD does away with this redundancy by performing one update at a time. It is therefore usually much faster and can also be used to learn online. SGD performs frequent updates with a high vari-

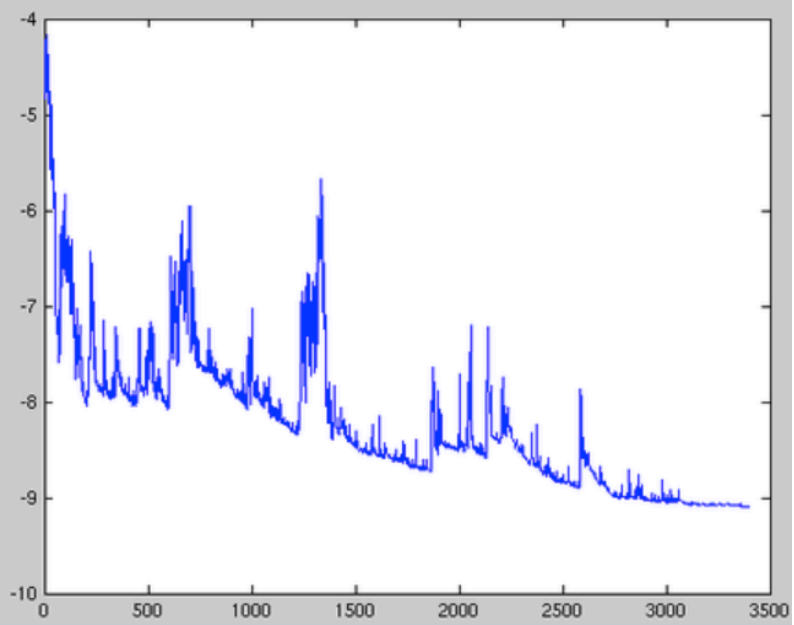


Fig. 2.11: SGD fluctuation. Figure is taken from Wikipedia

ance that cause the objective function to fluctuate heavily as in Figure 2.11. While batch gradient descent converges to the minimum of the basin the parameters are placed in, SGD's fluctuation, on the one hand, enables it to jump to new and potentially better local minima. On the other hand, this ultimately complicates convergence to the exact minimum, as SGD will keep overshooting. However, it has been shown that when we slowly decrease the learning rate, SGD shows the same convergence behavior as batch gradient descent (use a batch of samples in one iteration), almost certainly converging to a local or the global minimum for non-convex and convex optimization respectively.

Momentum

Further proposals include the momentum method, which appeared in Rumelhart, Hinton and Williams' seminal paper on backpropagation learning [27]. Stochastic gradient descent with momentum remembers the update Δw at each iteration, and determines the

next update as a linear combination of the gradient and the previous update [34]:

$$\Delta w := \alpha \Delta w - \eta \nabla Q_i(w) \tag{2.51}$$

$$w := w + \Delta w \tag{2.52}$$

that leads to:

$$w := w - \eta \nabla Q_i(w) + \alpha \Delta w \tag{2.53}$$

where the parameter w which minimizes $Q(w)$ is to be estimated, and η is a step size (sometimes called the learning rate in machine learning)

The name momentum stems from an analogy to momentum in physics: the weight vector w, thought of as a particle traveling through parameter space [27], incurs acceleration from the gradient of the loss ("force"). Unlike in classical stochastic gradient descent, it tends to keep traveling in the same direction, preventing oscillations. Momentum has been used successfully for various cases. Figure 2.12 describe the training process between SGD and SGD with momentum.

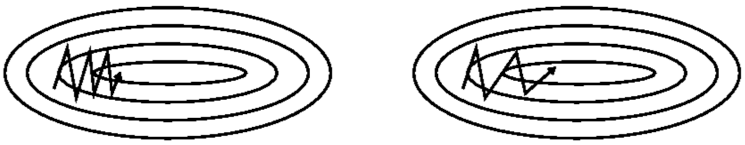


Fig. 2.12: Trainging process between SGD (left), and SGD with momentum (right).

AdaGrad

AdaGrad (for adaptive gradient algorithm) is a modified stochastic gradient descent with per-parameter learning rate, first published in 2011 [35], [36]. Informally, this increases the learning rate for more sparse parameters and decreases the learning rate for less sparse ones. This strategy often improves convergence performance over standard stochastic gradient descent in settings where data is sparse and sparse parameters are more informative. Examples of such applications include natural language processing and image recognition [35]. It still has a base learning rate η , but this is multiplied with the elements of a vector $G_{j,j}$ which is the diagonal of the outer product matrix.

$$G = \sum_{\tau=1}^{t} g_\tau g_\tau^\mathsf{T} \tag{2.54}$$

where $g_\tau = \nabla Q_i(w)$, the gradient, at iteration τ. The diagonal is given by the following:

$$G_{j,j} = \sum_{\tau=1}^{t} g_{\tau,j}^2 \tag{2.55}$$

This vector is updated after every iteration. The formula for an update is now become:

$$w := w - \eta \, \mathrm{diag}(G)^{-\frac{1}{2}} \circ g \tag{2.56}$$

or, can be writted as per-parameter updates:

$$w_j := w_j - \frac{\eta}{\sqrt{G_{j,j}}} g_j. \tag{2.57}$$

Each $G_{(i,i)}$ gives rise to a scaling factor for the learning rate that applies to a single parameter wi. Since the denominator in this factor, $\sqrt{G_i} = \sqrt{\sum_{\tau=1}^{t} g_\tau^2}$ is the L2 norm of previous derivatives, extreme parameter updates get dampened, while parameters that get few or small updates receive higher learning rates [37].

RMSProp

RMSProp (for Root Mean Square Propagation) is also a method in which the learning rate is adapted for each of the parameters. The idea is to divide the learning rate for a weight by a running average of the magnitudes of recent gradients for that weight [38]. So, first the running average is calculated in terms of means square :

$$v(w,t) := \gamma v(w,t-1) + (1-\gamma)(\nabla Q_i(w))^2 \tag{2.58}$$

where, γ is the forgetting factor. And the parameters are updated as the following:

$$w := w - \frac{\eta}{\sqrt{v(w,t)}} \nabla Q_i(w) \tag{2.59}$$

RMSProp has shown excellent adaptation of learning rate in different applications.

Adam

Adam (short for Adaptive Moment Estimation) [39] is an update to the RMSProp optimizer. In this optimization algorithm, running averages of both the gradients and the second moments of the gradients are used. Given parameters $w^{(t)}$ and a loss function $L^{(t)}$, where t indexes the current training iteration (indexed at 1), Adam's parameter update is given by the following:

$$m_w^{(t+1)} \leftarrow \beta_1 m_w^{(t)} + (1-\beta_1)\nabla_w L^{(t)} \tag{2.60}$$

$$v_w^{(t+1)} \leftarrow \beta_2 v_w^{(t)} + (1-\beta_2)(\nabla_w L^{(t)})^2 \tag{2.61}$$

$$\hat{m}_w = \frac{m_w^{(t+1)}}{1-\beta_1^t} \hat{m}_w = \frac{m_w^{(t+1)}}{1-\beta_1^t} \tag{2.62}$$

$$\hat{v}_w = \frac{v_w^{(t+1)}}{1-\beta_2^t} \hat{v}_w = \frac{v_w^{(t+1)}}{1-\beta_2^t} \tag{2.63}$$

$$w^{(t+1)} \leftarrow w^{(t)} - \eta \frac{\hat{m}_w}{\sqrt{\hat{v}_w} + \epsilon} \tag{2.64}$$

2.4 Generative Adversarial Networks

Generative adversarial networks (GANs) are a class of artificial intelligence algorithms used in unsupervised machine learning, implemented by a system of two neural networks contesting with each other in a zero-sum game framework. GANs are generative models devised by Goodfellow et al. [22] in 2014. In a GAN setup, two differentiable functions, represented by neural networks, are locked in a game. The two players (the generator G and the discriminator D) have different roles in this framework. Figure 2.13 shows the framework of a basic Generative Adversarial Network (GAN). A normal GAN consist of two sub network, G and D. G takes the input as a random noise. The noise is allowed to be sampled from any random distribution such as Gaussian or Unifrom distribution. The output of G is fake data, in our case, image. On the other hand, D will take the

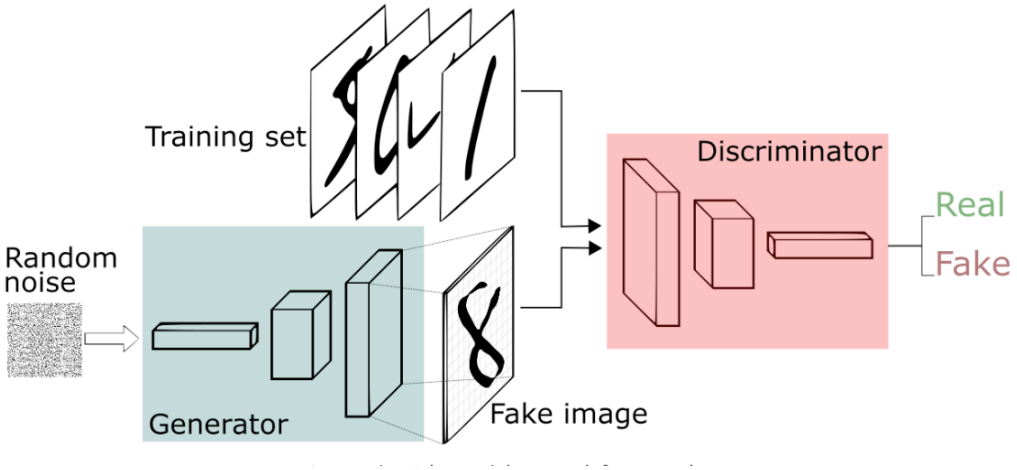


Fig. 2.13: Generative Adversarial Network framework.

input as images, and output prediction on distinguishing this input is real one or a fake one. The generator tries to produce data that come from some probability distribution. That would be you trying to reproduce the party's tickets. The discriminator acts like a judge. It gets to decide if the input comes from the generator or from the true training set. That would be the party's security comparing your fake ticket with the true ticket to find flaws in your design.

In summary, the game follows with:

1. The generator trying to maximize the probability of making the discriminator mistakes its inputs as real.
2. The discriminator guiding the generator to produce more realistic images.

The training process itself will follow the belowing minimax game with network G and D:

$$\min_G \max_D \mathbb{E}_{x\sim p_X(x)}[logD(x)] + \mathbb{E}_{z\sim p_Z(z)}[log(1 - D(G(z)))] \tag{2.65}$$

The generator G implicitly defines a probability distribution p_g as the distribution of the samples generated by the generator G when the input z is sample from distribution p_z. Therefore, we can prove that when at the global minimal of Equation (2.65), we can have the distribution modeled by G is exactly the same distribution of our training data p_{data}. The proof is followed with the material given in the original GAN paper [22].

Theorem 1: At the global minimal of Equation (2.65), $p_g = p_{data}$ We first consider the optimal discriminator D for any given generator G.

Propostion 1: For G fixed, the optimal discriminator D is:

$$D^*_G(x) = \frac{p_{data}(x)}{p_{data}(x) + p_g(x)} \tag{2.66}$$

Proof. The training criterion for the discriminator D, given any generator G, is to maximize the quantity $V(G, D)$:

$$V(G,D) = \int_x p_{data}(x)log(D(x))dx + \int_z p_z(z)log(1 - D(G(z)))dz \tag{2.67}$$
$$= \int_x p_{data}(x)log(D(x)) + p_g(x)log(1 - D(x))dx$$

For any $(a, b) \in \mathbb{R}^2$
$\{0, 0\}$, the function $y \to alog(y) + blog(1 - y)$ achieves its maximum in $[0, 1]$ at $\frac{a}{a+b}$. □

Note that the training objective for D can be interpreted as maximizing the log-likelihood for estimating the condional probability $P(Y = y|x)$, where Y indicates whether x comes from p_{data} (with $y = 1$) or from p_g ($y = 0$). The minimax game in Equation (2.65) can now be reformulated as the following:

$$C(G) = \max_D V(G, D) \tag{2.68}$$
$$= \mathbb{E}_{x\sim p_{data}}[logD^*_G(x)] + \mathbb{E}_{z\sim p_z}[log(1 - D^*_G(G(z)))] \tag{2.69}$$
$$= \mathbb{E}_{x\sim p_{data}}[logD^*_G(x)] + \mathbb{E}_{x\sim p_g}[log(1 - D^*_G(x))] \tag{2.70}$$
$$= \mathbb{E}_{x\sim p_{data}}[log\frac{p_{data}(x)}{p_{data}(x) + p_g(x)}] + \mathbb{E}_{x\sim p_g}[log\frac{p_g(x)}{p_{data}(x) + p_g(x)}] \tag{2.71}$$
$$\tag{2.72}$$

Now we can prove with **Theorem 1**

Proof. For $p_g = p_{data}$, $D^*_G(x) = \frac{1}{2}$(consider in Equation (2.66)). Hence, by inspecting Equation (2.68) at $D^*_G(x) = \frac{1}{2}$, we find $C(G) = log\frac{1}{2} + log\frac{1}{2} = -log4$. To see that this the best possible value of $C(G)$, reached only for $p_g = p_{data}$, observe that:

$$\mathbb{E}_{x\sim p_{data}}[-log2] + \mathbb{E}_{x\sim p_g}[-log2] = -log4$$

and that by subtracting this expression from $C(G) = V(D^*_G, G)$, we obtain:

$$C(G) = -log(4) + KL(p_{data}||\frac{p_{data} + p_g}{2}) + KL(p_g||\frac{p_{data} + p_g}{2}) \tag{2.73}$$

where KL is the Kullback-Leibler divergence. We recognize in the previous expression the Jensen-Shannon divergence between the model's distribution and the data generating process :

$$C(G) = -log(4) + 2 \times JSD(p_{data}||p_g) \tag{2.74}$$

Since the Jensen-Shannon divergence between two distributions is always non-negative and zero only when they are equal, we have shown that $C^* = -log(4)$ is the global minimum of $C(G)$ and that the only solution is $p_g = p_{data}$. *i.e.*, the generative model perfectly replicating the data generating process. □

Theoretically, in the perfect Nash equilibrium [40], the generator would capture the general training data distribution. As a result, the discriminator would always be unsure of whether its inputs are real or not.

For better improve the learning ability of GAN, Deep Convolutional Generative Adversarial Network (DCGAN) [8] has been proposed. Figure 2.14 shows the architecture of generator G in DCGAN. In DCGAN paper the authors describe the combination of some deep learning techniques as key for training GANs. These techniques include:

1. Use the all convolutional layer
2. Use Batch Normalization (BN).

The first emphasizes strided convolutions (instead of pooling layers) for both: increasing and decreasing feature's spatial dimensions. And the second normalizes the feature vectors to have zero mean and unit variance in all layers. This helps to stabilize learning and to deal with poor weight initialization problems. Notice that in G, deconvolutional layers

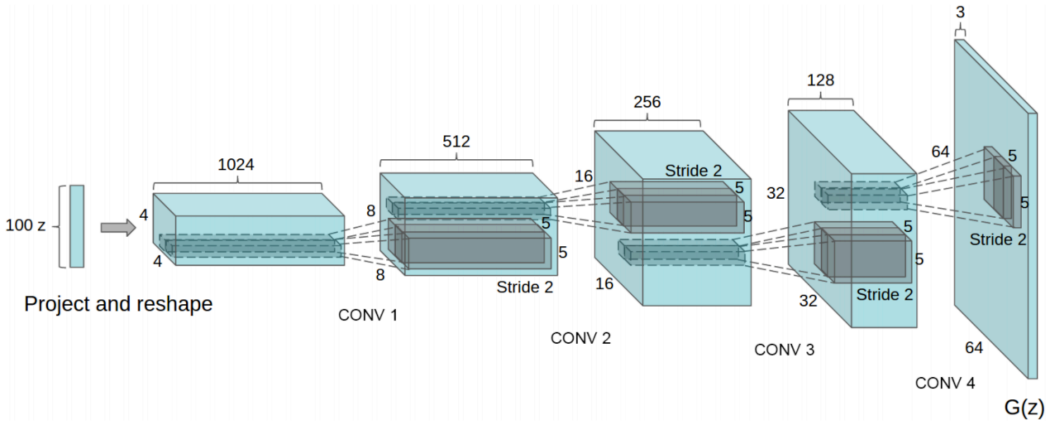


Fig. 2.14: Adapted from the DCGAN paper [8]. The Generator network implemented here. Note the non-existence of fully connected and pooling layers.

are used to "up sample" the feature maps. Use the all convolutional layer (along with deconvolutional layer) favours the learning in GAN based model. Our implementation also based on DCGAN's structure, see in Chapter 4 for more details.

Chapter 3

Lantent Space Understandable Generative Adversarial Netork

This chapter describes a latent space understandable network: Self-excited Generative Adversarial Network (SelfExGAN), a novel self-excited structure based on adversarial learning. Compared with the conventional generative adversarial networks, SelfExGAN consists of three components, which are encoder (E), generator (G), and discriminator (D). Different from other papers which directly apply reconstruction loss between encoder and generator, SelfExGAN introduces Nash equilibrium [40] between these three parts in order to discover the correspondence between latent inputs and training data spontaneously. Compared with other GAN based networks only take latent variable z as random input, SelfExGAN is trained to discover the "meaning" behind z. SelfExGAN is able to determine the correspondence between latent input space and training data space. SelfExGAN is trained in a self-excited way. That is, E takes the input from G's output, G takes the input sampled from E's output. E and G together constitute a positive feedback in SelfExGAN. The most attractive point of SelfExGAN is that it can use the learned correspondence to guide G to generate homomorphic samples given different latent inputs in an unsupervised learning manner. We demonstrate the learning ability and as well as its applications in Chapter 4.

3.1 Introduction and Intuition

Generative Adversarial Networks (GANs) [22], are neural networks that learn to create new samples similar to the training data. A GAN is composed of two sub-neural networks which are called generator (G) and discriminator (D) respectively. The main idea behind a GAN is to have these two competing neural network models trying to achieve opposite objectives. Generator takes random noise as input and tries to generate samples with high authenticity. On the other hand, discriminator receives samples from both the generator and the training data, and has to be able to distinguish between these two sources. G and D play a continuous game, where the generator is learning to produce more realistic samples, and the discriminator is learning to distinguish generated data from real one. These two networks are trained simultaneously, and the objective is that the competition will drive the generated samples to be indistinguishable from real ones.

GANs are a powerful approach to unsupervised and semi-supervised learning where the goal is to discover the hidden structure within data without relying on external labels. Traditional Machine Learning (ML) is mostly discriminative – the goal being to discover a map from inputs to outputs, like images to class names presented in it. GANs, on the other hand, learn in a different way. They try to recreate the rich imagery of the world

in which they will be used. Instead of discriminating the inputs, they try to replicate the (hidden) statistical process that is behind the data being observed. They start by generating "hypothesis" that become more realistic and plausible as the learning process evolves. Thus, these models have more powerful expression capabilities as they explore regulation in data and try to reason about the structure and possibilities of the world that are consistent with given observations.

The goal of training GANs is to find a Nash equilibrium between two players G and D. The optimization of Equation (3.5) essentially has no closed form, unlike standard loss functions like log-loss or squared error. Unfortunately, finding a Nash equilibrium in GANs remains remarkable difficult to train, we are not aware of any algorithms exists guarantee that the training of GAN can be converged [41],[8]. Thus, optimizing this loss function is very hard and requires a lot of trial-and-error regarding the network structure and training protocol.

Based on the idea of data creation, GANs can compensate the limitation of current machine learning technologies and be used in various applications. Including density estimation, image denoising (creating high-quality images from low resolution or noisy ones) and inpainting (recovering the whole image after a partial occlusion), data compression, scene understanding, representation learning, 3D scene construction, video generation, semi-supervised classification or even hierarchical control [42] [43] [44] [45].

Even though the expectation for GANs is high, there is one fatal issue needs to be improved which is the "model collapse" [46]. In such a scenario, generator will exhibit very poor diversity among generated samples regardless of the variations in the latent input which limits the learning ability when training GANs. Several recent papers introduced various sophisticated methods which focus on improving the stability of training the GANs. Salimans *et al.* proposed five useful techniques to encourage the divergent creation, which are feature matching, batch discrimination, historical averaging, one-sided label smoothing and virtual batch normalization [41]. Mirza*et al.* and Odena *et al.* proposed to condition both generator and discriminator of GANs on side information to perform category conditioned image generation. The labels of training data is used as extra information to guide the training [47] [48]. Besides, some papers indicated particular architecture bless the training as well. Radford *et al.* replaced pooling layers with strided convolutions, removed fully connected hidden layers for deeper architectures, and used LeakyReLU activation instead of ReLU activation [8]. Isola *et al.* proposed a novel structure with skip connection in generator called "U-Net" to let the useful information directly flow across the net [49]. Moreover, how to design the training processing also affects the divergence of generated data, *e.g.* the excessive optimal D will hinder G learn better [50].

From our experience, other methods could be using specific optimiser for specific case, *e.g.* use SGD for D and Adam [39] for G; avoid sparse gradients *e.g.* do not use max pool or ReLU; during every iteration, update G twice,then update D, we find in order to get better results, we cannot train D too well at each time. Because the excessive optimal D will hinder G learn better [51].

Compared with other generative model *e.g.* Bayesian-based model, GANs do not formulate a prior distribution for observing new training data. Specifically, GANs only require any random latent inputs without any assumption, and G is trained to learn the mapping from z to "fake" samples. We argue that this could be the most attractive point of GANs, but also the reason behind of the unstable training problem. Because there is no intervene path between z and the generator, sometimes very different z vectors will produce nearly the same generated data, even though G still can accomplish its goal. *i.e.* G trends not to consider the data diversity, it starts to generate a few samples which are the most similar ones to training data. In such case, G will fall in an aimless learning and incur producing

only a minimal number of unique examples similar to training data.

GANs sometimes map very different z into exactly same sample which inevitably incurs training falls in a aimless learning; ie generator will not become better during training, the generated sample will lose divergence. It's called "model collapse". [50] and [52] present a thorough mathematic analysis that proves in original GANs [22] the probability of fake data manifold learned by G and true data manifold match on a big part of the space is identically equal to 0. If there always exists an optimal D can distinguish fake data and true data, JS divergence based GANs will never converge. According to their mathematic induction, they demonstrate a more power generative model called Wasserstein GAN.

In this paper, we propose a latent space understandable network – Self-excited Generative Adversarial Network (SelfExGAN), which not only learns how to generate new samples, but also determines the correspondence between the latent inputs and the training data. In addition, SelfExGAN is able to use the learned correspondence to guide G to generate homomorphic sample given different latent input.

On another view point, this paper presents a new way to fix "model collapse". We present SelfExGAN not only learns to generate new samples but also learns to discriminate the characteristics in training data. SelfExGAN does not try to directly map a latent input z vector into arbitrary generated data, but learns to map different z vectors to specific different generated samples which are similar to training data.

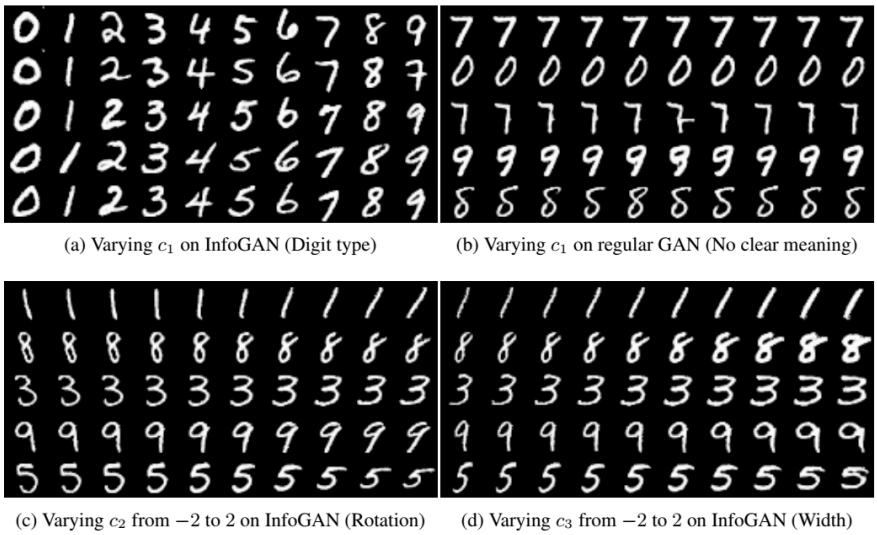
We propose a SelfExGAN which can emulate the meaning of z by itself in a completely unsupervised learning manner; *i.e.* SelfExGAN can think for itself, during training, it can understand what kind of z will lead to generate what kind of fake data. Because of that, the learned fake data distribution by G can be easily dragged close to true data distribution.

3.2 Related Works

Our idea is mainly inspired by [10] and [11]. Concretely speaking, [10] presented several techniques for sampling and visualizing the latent spaces of generative models involves both Variational Autoencoders (VAEs) [53] and GANs. In [10], the authors introduce several techniques for sampling and visualizing the latent spaces of generative models. Such as: replacing linear interpolation with spherical linear interpolation prevents diverging from a model's prior distribution and produces sharper samples. For example, a vector can be computed which represents the smile attribute, which by shorthand we call a smile vector. Following (Larsen et al. [54]), the smile vector can be computed by simply subtracting the mean vector for images without the smile attribute from the mean vector for images with the smile attribute. This smile vector can then be applied to in a positive or negative direction to manipulate this visual attribute on samples taken from latent space. Figure 3.1 demonstrate the results of traversals along the generated results from a smile vector.

Fig. 3.1: Traversals along the smile vector. Train on CelebA dataset [9]. Quote from original paper [10].

[11] adds some independent random variables with z learning together, and they found varying those variables in the latent space will have the specific meaning when observing generated data in several datasets. Figure 3.2 demonstrates when varying the additional dimension in the latent space, the generated output digital number changes accordingly. For example, changing on the c_1 dimension, the digital type will change accordingly. From their results, we realize that we can make the model learn the correspondence between the latent space Ω_Z and real data space Ω_X.



(a) Varying c_1 on InfoGAN (Digit type) (b) Varying c_1 on regular GAN (No clear meaning)

(c) Varying c_2 from -2 to 2 on InfoGAN (Rotation) (d) Varying c_3 from -2 to 2 on InfoGAN (Width)

Fig. 3.2: Demonstration on changing independent addtional dimension. When changing the addtional dimension on latent space, the generated sample will change accordingly. Quote from original paper [11].

Other papers such as Conditional GAN [47], Stack GAN [55] use label as extra information to training the Generative Adversarial Network for finding the correspondence between the hidden space and data space. Conditional GAN uses the label of belonging classes to generate fake image with a specific label. Stack GAN uses the label of corresponding text to generate according photos.

Works like [54] combine the model of GAN with VAE to discover the hidden space. Figure 3.3 shows the basic structure of this idea. The discussion of VAE is **out of range** in this thesis.

Consider on the architecture, the most similar work to us is proposed in paper Adversarial Feature Learning by Jeff Donahue et al. [12]. They named their model by BiGAN. Figure 3.4 shows the structure of BiGAN. BiGAN also consists of three parts which are encoder, generator, and discriminator. The difference compared with our works is, the discriminator in BiGAN is to try to distinguish the input comes from the generator or the encoder. In our proposed model, discriminator tries to distinguish the fake pair with the real pair. See next section for more details. Another very different point is that in BiGAN, no discriminate judge on the fake samples, they directly introduce a reconstruction loss for the data input. On the other hand, BiGAN aims to follow this adversarial training for better feature extraction. Figure 3.5 shows the training results on different convolutional filter of three modules E, G, D in BiGAN. Table 3.1 demonstrates the learning ability on adversarial training with convolution layers.

In paper IVE-GAN [13], they use a similar structure with BiGAN. Figure 3.6 shows

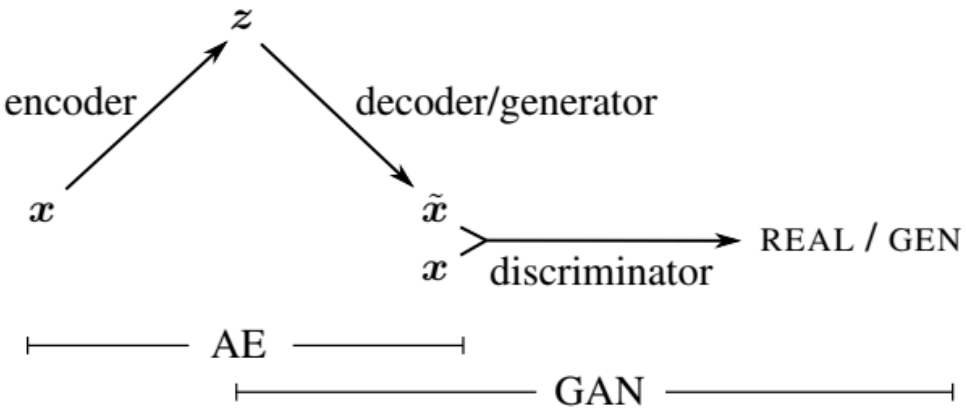


Fig. 3.3: Basic structure of combing GAN with VAE.

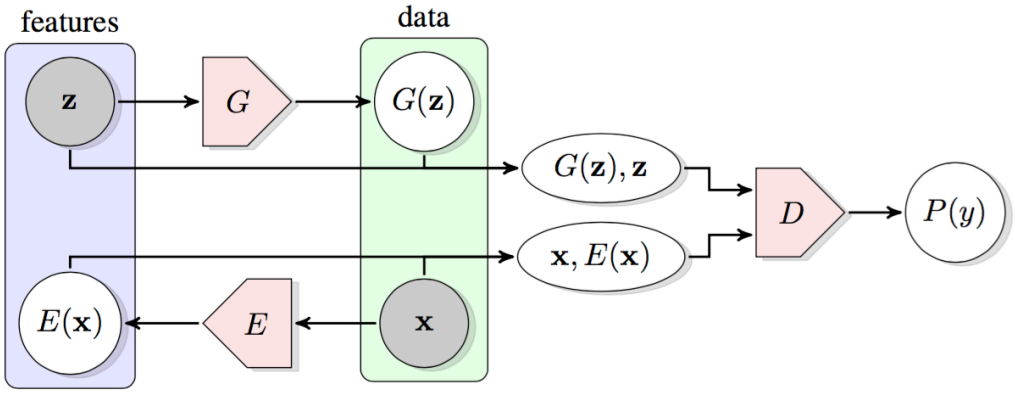


Fig. 3.4: The structure of BiGAN. Quote from original paper [12]

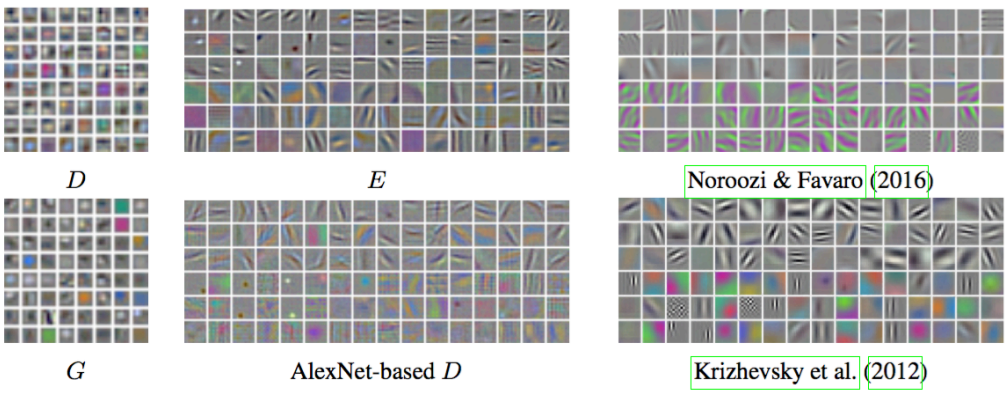


Fig. 3.5: The convolutional filter learned by E, G, D, in BiGAN. Quote from original paper [12]

Table 3.1: Classification accuracy of BiGAN for Imagenet dataset[16] compare with other methods.

Methods	conv1	conv2	conv3	conv4	conv5
Noroozi et al. [56]	48.5	41.0	34.8	27.1	12.0
Wang et al. [57]	51.8	46.9	42.8	38.8	29.8
Doersh et al. [58]	53.1	47.6	48.7	45.6	30.4
BiGAN	56.2	54.4	49.4	43.9	33.3

the architecture of IVE-GAN. They introduce another sub net named "translator" for making the discriminator easy to distinguish the pair relation. Figure 3.7 visualizes the latent space representation results with CelebA dataset.

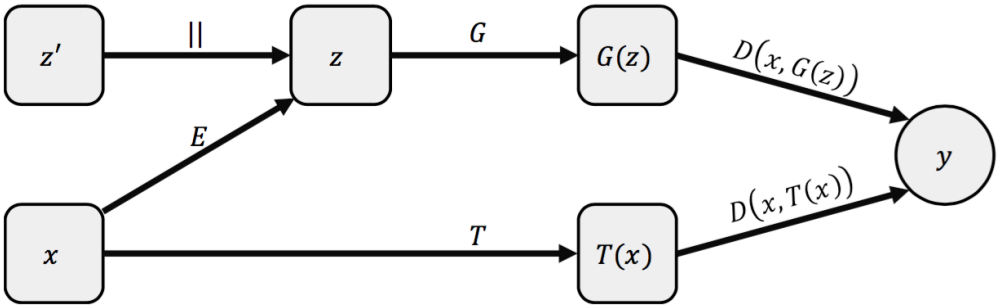


Fig. 3.6: Illustration of the IVE-GAN architecture. Quote from original paper [13]

Dmitry Ulyanov et al. [15] present a new autoencoder-type architecture that is trainable in an unsupervised mode, sustains both generation and inference, and has the quality of conditional and unconditional samples boosted by adversarial learning. Unlike previous hybrids of autoencoders and adversarial networks, the adversarial game in their approach is set up directly between the encoder and the generator. Figure 3.8 presents the idea in their approach which are use ecoder to build the bridge between the latent space and data space. They mention in their paper that on external mapping is used during training. Actually, in their official implementation, they do add reconstruction loss as bellow in their code:

$$L_X(G_\theta, E_\psi) = \mathbb{E}_{x\sim X}||x - G_\theta(E_\psi(x))||_1 \tag{3.1}$$
$$L_Z(G_\theta, E_\psi) = \mathbb{E}_{z\sim Z}||z - E_\psi(G_\theta(z))||_2 \tag{3.2}$$

3.3 Self-excited Generative Adversarial Network

SelfExGAN consists of three components, which are encoder (E), generator (G), and discriminator (D). The overall model is depicted in Fig. 3.9. At first sight, the proposed model seems a little common compared with existing models. Zhu *et al.* applied the similar structure to accomplish unpaired image mapping, called Cycle-Consistent Adversarial Networks [59]. Kim *et al.* introduced two pairs of generators with sharing parameters as two pairs of autoencoders serving as mapping from domain A to domain B and vice versa [60]. Jeff *et al.* proposed a learning of the inverse mapping in GANs, named Bidirectional

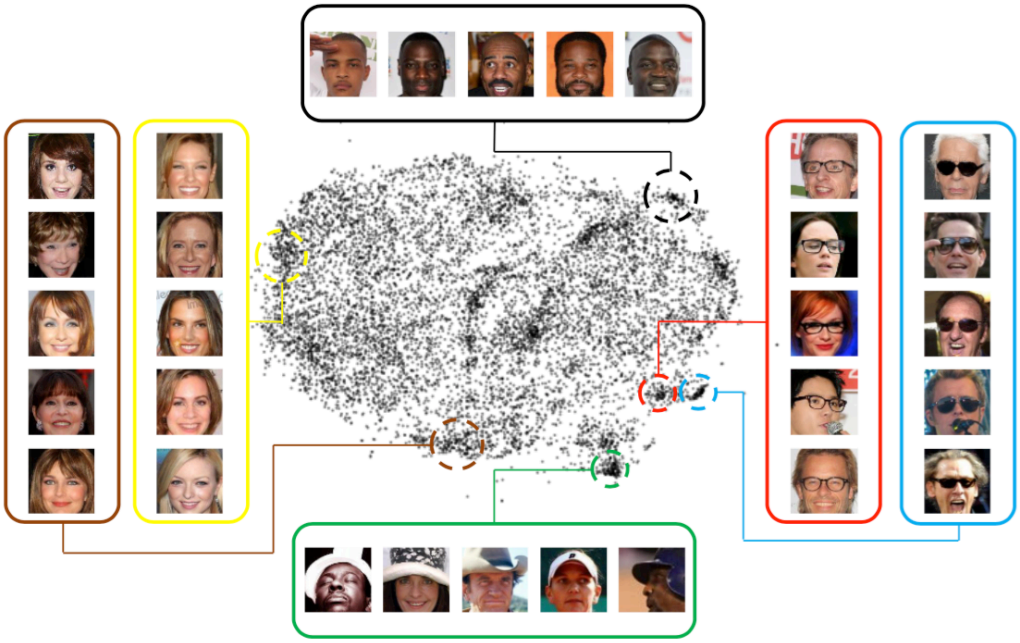


Fig. 3.7: Visualization of the 2-dimensional t-SNE [14] of the latent space representation. Quote from original paper [13]

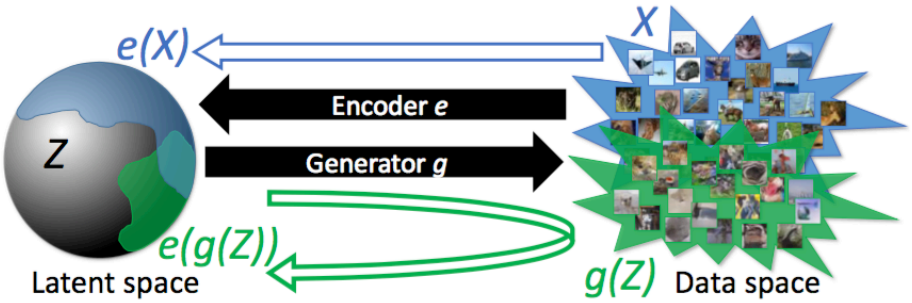


Fig. 3.8: Idea of AGE. Green region: modelled by G; Blue region: modelled by E; Gray region: latent space; Objective: blue overlap with green, and be full of gray space. Quote from original paper [15]

Generative Adversarial Networks as a means of better feature learning and representation [12].

Our proposed model is distinct from existing models mainly in three aspects:
(1) We **do not** apply reconstruction loss between encoder and generator, instead of that, we introduce another adversarial loss between E, G, and D.
(2) The meaning of latent input will be learned by SelfExGAN in an adversarial and self-excited way, **not** directly sampled from a fix distribution such as Uniform or Gaussian distribution. On the other hand, we introduce a matching loss for E to output a random distribution, see Chapter 4 for more details.
(3) There is **no** sharing parameters between E and G. During training, the parameters of both E and G will be updated separately.

Fig. 3.9 describes our proposed model Self-excited Generative Adversarial Network (SelfExGAN). The most distinctive thing is SelfExGAN no longer takes the input as the

random meaningless z, but learn the correspondence between latent input and training data. In Section 3.3.1 we will give a detail explanation on how SelfExGAN works. Mathematic intuition will be demonstrated in Section 3.3.2. In Section 3.3.3 we view our work in a different way, compare it with Compressive Sensing [61].

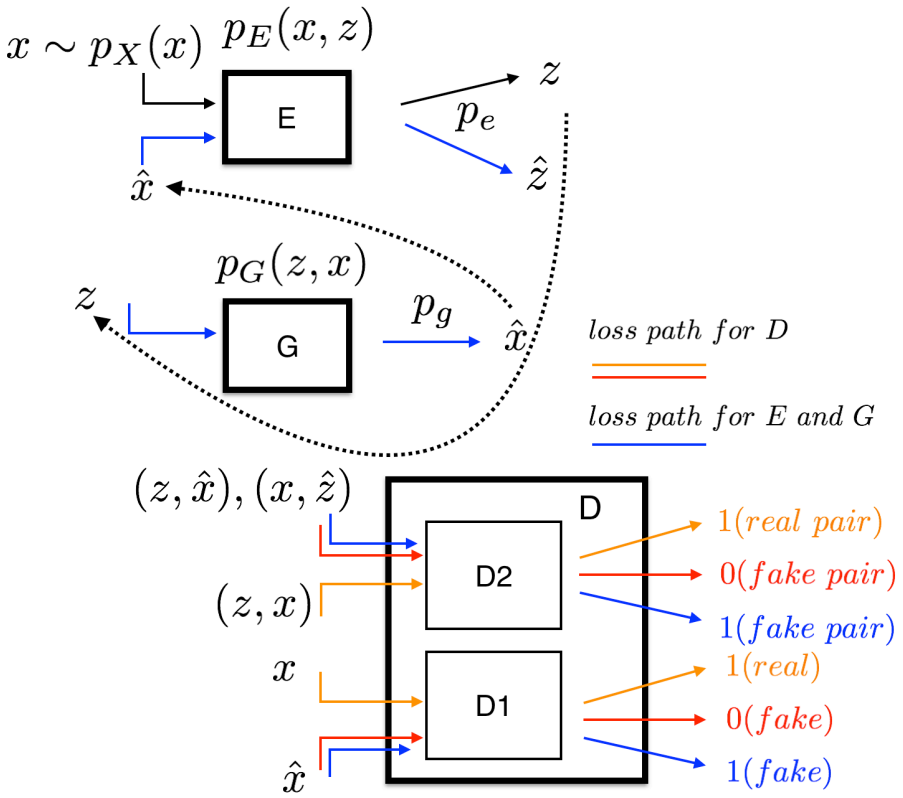


Fig. 3.9: The structure of SelfExGAN. Our model contains of three parts which are E, G, and D. $p_E(x,z)$ denotes the joint distribution learned by E, $p_G(z,x)$ denotes the joint distribution learned by G. $p_g(x)$ denotes the distribution learned by G for mapping z into x, $p_e(z)$ denotes the distribution learned by E for mapping x to z. $D1$ trying to distinguish the generated $\hat{x}$ and real x, $D2$ trying to distinguish the fake pair $(z,\hat{x})$, $(x,\hat{z})$, and real pair (z,x).

3.3.1 Structure of SelfExGAN

SelfExGAN contains of three parts which are E, G, and D. We denotes the real data distribution as p_X, x is the training data sampled from p_X. During training, E models a joint distribution between x and latent variable z denoted as $p_E(x,z)$. Likewise, G induces a joint distribution $p_G(z,x)$. $p_e(z)$ denotes the distribution learned by E mapping data x into z, likewise for $p_g(x)$ (see Figure 3.9).

Different from conventional GANs, SelfExGAN on longer takes the latent inputs as random and meaningless information. That is, during training, the input of G will be sampled from the output of E. There exist two groups of adversarial learning in SelfExGAN. One is the same with general GANs, discriminator $D1$ is trained to separate generated data $\hat{x}$ and real data x. Another adversarial learning is induced in SelfExGAN, $D2$ is trained to differentiate the fake pair $(z,\hat{x})$, $(x,\hat{z})$ from real pair (z,x) (Because both $\hat{x}$ and $\hat{z}$ come from generator G, we consider those fake pairs, see Fig. 3.9). In SelfExGAN, encoder E

discovers the corresponding low dimensional latent space representation for the original data x in high dimensional data space Ω_X. In next section, we will prove that based on this structure the generated data $\hat{x}$ is the corresponding sample in Ω_X given the particular z mathematically.

It is worthy to mention that SelfExGAN will be trained in a self-excited fashion. That is, along with training, E will produce better z for representing corresponding x; this results in G generating more plausible sample given that z. Then, E will take the more real sample generated by G as a new training input. Thus, E and G will form a positive feedback in a self-excited way. According to adversarial learning, when Nash equilibrium achieved between E, G, and D, D will no longer be able to distinguish the generated data with real one and generated pair with real pair. Namely, given specific latent input z, SelfExGAN will precisely generate the corresponding sample reckoning by itself (see Chapter 4). We argue that it is important for the model to learn that given different z what kind of sample is the most suitable one to generate. Otherwise, G tends to do aimless learning, the generated samples will keep losing divergence; *i.e.* even far distinguished z vector may produce the nearly identical sample.

3.3.2 Theoretical Results

In SelfExGAN, we have not only one adversarial training involves between G and D, but with another one between E, G, and D. The latent input z will be determined by E which additionally train to discover a bijection relationship between space Ω_X and Ω_Z. The discriminator in SelfExGAN will have two group of goals: one is output 1 if the input is real data, 0 for fake one; the other is output 1 if the input is real pair, 0 for fake pair. Next, we will prove based on the above structure we are able to achieve learning the correspondence between space Ω_X and Ω_Z.

According to our description in the former section, the objective of SelfExGAN can be defined as a min-max learning on value function $V(G, E, D)$:

$$\min_{G,E} \max_{D} V(G, E, D) \tag{3.3}$$

where

$$\begin{aligned} &V(G, E, D) = \\ &\quad \mathbb{E}_{(x,z)\sim p_E(\cdot,x)}[log(D(x, z)]+ \\ &\quad \mathbb{E}_{(x,z)\sim p_G(\cdot,z)}[log(1 - (D(x, z))] \end{aligned} \tag{3.4}$$

We optimize this min-max objective using the same alternating moment based optimization. See Chapter 4 for implementation detail. Fig. 3.9 gives a clear description of Equation (3.4). If we consider $p_e(z)$ as one kind of prior distribution $p_Z(z)$ assuming for z, then the objective of SelfExGAN degenerates into Equation (3.5):

$$\begin{aligned} &\min_{G} \max_{D} \mathbb{E}_{x\sim p_X(x)}[logD(x)]+ \\ &\quad \mathbb{E}_{z\sim p_Z(z)}[log(1 - D(G(z)))] \end{aligned} \tag{3.5}$$

Goodfellow *et al.* [22] proved the optimal discriminator has the shape of:

$$D^* = \frac{p_X(x)}{p_X(x) + p_g(x)}$$

and that $V(G, D^*) = 2D_{JS}(p_X(x)||p_g(x)) - 2log2$ (see **Definitions**), so min-maxing Equation (3.3) still guarantee that $p_X(x) \simeq p_g(x)$. For indicating SelfExGAN is able to discover

the correspondence between space Ω_X and Ω_Z, next, we will prove that if and only if at global minimum $p_G(x,z) \simeq p_E(x,z)$ and that must exists a bijection pair between Ω_X and Ω_Z; *i.e.* $G(E(x)) = x$ and vice versa.

Definitions : Let $p_G(x,z) = p_{GZ}(x|z)p_Z(z)$ and $p_E(x,z) = p_{EX}(z|x)p_X(x)$ be the joint distribution modelled by G and E respectively, $\Omega = \Omega_X \times \Omega_Z$ be the joint latent and data space. We assume both G and E are deterministic functions; *i.e.* with conditionals $p_{GZ}(x|z) = \delta(x - G(z))$ and $p_{EX}(z|x) = \delta(z - E(x))$ defined as δ functions. Then the measure for a region $R \subseteq \Omega$ is:

$$
\begin{aligned}
P_E(R) &= \int_\Omega p_E(x,z)\mathbf{1}_{[(x,z)\in R]}d(x,z) \\
&= \int_{\Omega_X} p_X(x) \int_{\Omega_Z} p_{EX}(z|x)\mathbf{1}_{[(x,z)\in R]}dzdx \\
&= \int_{\Omega_X} p_X(x)(\int_{\Omega_Z} \delta(z - E(x))\mathbf{1}_{[(x,z)\in R]}dz)dx \\
&= \int_{\Omega_X} p_X(x)\mathbf{1}_{[(x,E(x))\in R]}dx \\
P_G(R) &= \int_\Omega p_G(x,z)\mathbf{1}_{[(x,z)\in R]}d(x,z) \\
&= \int_{\Omega_Z} p_Z(z) \int_{\Omega_X} p_{GZ}(x|z)\mathbf{1}_{[(x,z)\in R]}dxdz \\
&= \int_{\Omega_Z} p_Z(z)(\int_{\Omega_X} \delta(x - G(z))\mathbf{1}_{[(x,z)\in R]}dx)dz \\
&= \int_{\Omega_Z} p_Z(z)\mathbf{1}_{[(G(z),z)\in R]}dz
\end{aligned}
$$

$D_{KL}(P||Q)$ and $D_{JS}(P||Q)$ respectively denote the Kullback-Leibler (KL) and Jensen-Shannon (JS) divergences between two probability measure P and Q with the following definition:

$$
\begin{aligned}
D_{KL}(P||Q) &= \mathbb{E}_{x\sim p}[log(f_{PQ}(x)) \\
D_{JS}(P||Q) &= \frac{1}{2}[D_{KL}(P||\frac{P+Q}{2}) + D_{KL}(Q||\frac{P+Q}{2})]
\end{aligned}
$$

where $f_{PQ} = \frac{dP}{dQ}$ is the Radon-Nikodym (RN) derivative of measure P with respect to measure Q.

Proposition : For any E and G, the optimal discriminator $D^*_{EG} = argmax_D V(G,E,D)$ is the RN derivative $f_{EG} = \frac{dP_E}{d(P_E+P_G)}$ of measue P_E with respect to measure $P_E + P_G$.

Proof. For measures P and Q on space Ω, with P absolutely continuous with respect to Q, the RN derivative $f_{PQ} = \frac{dP}{dQ}$ exists, and we have:

$$
\begin{aligned}
\mathbb{E}_{x\sim p}[g(x)] &= \int_\Omega gdP = \int_\Omega g\frac{dP}{dQ}dQ \\
&= \int_\Omega gf_{PQ}dQ = \mathbb{E}_{x\sim q}[f_{PQ}(x)g(x)]
\end{aligned} \tag{3.6}
$$

Let the probability measures $P_{EG} = \frac{P_E+P_G}{2}$ denote the average of measures P_E and P_G. Both P_E and P_G are each absolutely continuous with respect to P_{EG}. Hence the RN derivatives $f_{EG} = \frac{dP_E}{d(P_E+P_G)} = \frac{1}{2}\frac{dP_E}{dP_{EG}}$ and $f_{GE} = \frac{dP_G}{d(P_E+P_G)} = \frac{1}{2}\frac{dP_G}{dP_{EG}}$ exist and sum to 1:

$$f_{EG} + f_{GE} = \frac{d(P_E + P_G)}{d(P_E + P_G)} = 1 \tag{3.7}$$

Then, use Equation (3.6), (3.7) we can rewrite Equation (3.3) as a single expectation under measure P_{EG} as following:

$$\begin{aligned}
&V(G,E,D)\\
&= \mathbb{E}_{(x,z)\sim p_E(\cdot,x)}[log(D(x,z)]+\\
&\quad \mathbb{E}_{(x,z)\sim p_G(\cdot,z)}[log(1-(D(x,z))]\\
&= \mathbb{E}_{(x,z)\sim p_{EG}}[2f_{EG}(x,z)logD(x,z)]+\\
&\quad \mathbb{E}_{(x,z)\sim p_{EG}}[2f_{GE}(x,z)log(1-(D(x,z)]\\
&= 2\mathbb{E}_{(x,z)\sim p_{EG}}[f_{EG}(x,z)logD(x,z)+\\
&\quad f_{GE}(x,z)log(1-D(x,z))]\\
&= 2\mathbb{E}_{(x,z)\sim p_{EG}}[f_{EG}(x,z)logD(x,z)+\\
&\quad (1-f_{EG}(x,z))log(1-D(x,z))]
\end{aligned}$$

Note that $argmax_y[xlog(y) + (1-x)log(1-y)] = x$ for any $x \in [0,1]$. Thus, $D^*_{EG} = f_{EG}$. □

Theorem 1: The global minimum of Equation (3.3) can be achieved if and only if $P_E = P_G$. At that point, the measure between P_E and P_G is $2log2$.

Proof. Using Propositon along with $1 - D^*_{EG} = 1 - f_{EG} = f_{GE}$, $P_{EG} = \frac{P_E+P_G}{2}$ (see in **Proposition**), we can rewrite Equation (3.3) as following:

$$\begin{aligned}
&V(G,E,D^*_{EG})\\
&= \mathbb{E}_{(x,z)\sim p_E(\cdot,x)}[log(D^*_{EG}(x,z)]+\\
&\quad \mathbb{E}_{(x,z)\sim p_G(\cdot,z)}[log(1-(D^*_{EG}(x,z))]\\
&= \mathbb{E}_{(x,z)\sim p_E(\cdot,x)}[log(f_{EG}(x,z)]+\\
&\quad \mathbb{E}_{(x,z)\sim p_G(\cdot,z)}[log(f_{GE}(x,z))]\\
&= \mathbb{E}_{(x,z)\sim p_E(\cdot,x)}[log(2f_{EG}(x,z)]+\\
&\quad \mathbb{E}_{(x,z)\sim p_G(\cdot,z)}[log(2f_{GE}(x,z))] - 2log2\\
&= D_{KL}(P_E||P_{EG}) + D_{KL}(P_G||P_{EG}) - 2log2\\
&= D_{KL}(P_E||\frac{P_E+P_G}{2}) + D_{KL}(P_G||\frac{P_E+P_G}{2}) - 2log2\\
&= 2D_{JS}(P_E||P_G) - 2log2
\end{aligned}$$

Thus, if and only if $P_E = P_G$ we can reach the global minimum, at that point the measure between P_E and P_G are $2log2$. □

This proof is similar to the proof in the basic GAN framework [22]. It is worthy to mention that, in SelfExGAN, we optimize a Jensen-Shannon divergence between a joint distribution involves both training data space Ω_X and latent space Ω_Z. Compared with

general GANs, this approach allows the proposed model to learn the representation of the original data in the low dimensional manifold Ω_Z. Next, we will give a mathematic explanation about the reason why we induce $D2$ in SelfExGAN; *i.e.* there must exist a bijection between Ω_X and Ω_Z.

Theorem 2: If the learned E and G are the optimal encoder and generator, then there must hold bijection between G and E almost everywhere. More concretely speaking, $G(E(x)) = x$ for all the $x \in \Omega_X$ and $E(G(z)) = z$ for all the $z \in \Omega_Z$.

Proof. Let $R_Z^0 = \{z \in \Omega_Z : z \neq E(G(z))\}$ be the region of Ω_Z where the bijection does not hold true. Let $R^0 = \{(x, z) \in \Omega : x = G(z) \wedge z \in R_Z^0\}$ be the region of Ω where $(G(z), z) \in R^0$ if and only $z \in R_Z^0$. Next, we will prove the measure of z on R_Z^0 is 0. Follow the same procedure, we can prove $R_X^0 = 0$ as well.
For proving **Theorem 2**, we will use the definitions of P_E and P_G for deterministic E and G from **Definitions**. The measure on region R_Z^0 is:

$$
\begin{aligned}
& P_Z(R_Z^0) \\
&= \int_{\Omega_Z} p_Z(z) \mathbf{1}_{[z \in R_Z^0]} dz \\
&= \int_{\Omega_Z} p_Z(z) \mathbf{1}_{[(G(z), z) \in R^0]} dz \\
&= P_G(R^0) = P_E(R^0) \\
&= \int_{\Omega_X} p_X(x) \mathbf{1}_{[(x, E(x)) \in R^0]} dx \\
&= \int_{\Omega_X} p_X(x) \underbrace{\mathbf{1}_{[x = G(E(x)) \wedge E(x) \in R_Z^0]}}_{E(G(E(x))) = E(x) \ and \ E(G(E(x))) \neq E(x)} dx \\
&= \int_{\Omega_X} p_X(x) 0 dx = 0
\end{aligned}
$$

Thus, there must a bijective mapping between x and z everywhere on Ω_X and Ω_Z. Account for that, we introduce a pair wise adversarial loss in SelfExGAN. □

Our experiment shows SelfExGAN is able to learn the relationship between the latent space and data space (see Chapter 4).

3.3.3 Related to Compressive Sensing

As we discussed in the previous sections, we argue that SelfExGAN is able to learn the correspondence between the latent space Ω_Z and Ω_X. Here we interpret that SelfExGAN can be considered as a Bayesian Compressive Sensing (CS) [62] approach for implying the high dimensional data can be reconstructed accurately in a low dimensional space. Our experiments show that SelfExGAN can generate prospective samples given the characteristic input data. CS theory relies on the empirical observation that signals (in our case, images) have a sparse representation in terms of a suitable basis. In SelfExGAN, encoder E serves as this functionality. Roughly speaking, learning process in SelfExGAN has a Bayesian perspective. For iteration t, we have a prior belief (determined in iteration $t-1$) that the basis learned by E should be able to represent input data in a sparse form (on Ω_Z space). And the training objective of SelfExGAN is to provide a posterior belief for sampling z from the learned representation by E, G can generate more real "fake" data to

"fool" discriminator D. During updating the parameter in each iteration, we estimate the underlying generated "fake" data with an "error bars" offering by D in SelfExGAN. In SelfExGAN, we induce adversarial loss instead of normal reconstruction loss. Specifically, adversarial learning between E, G, and D mimics a reconstruction loss with ℓ_0 norm.

Follow the **Definitions**, a KL divergence term of JS divergence in **Theorem 1** is:

$$\begin{aligned} &D_{KL}(P_E||\frac{P_E+P_G}{2}) = log2 + \int_\Omega log\frac{dP_E}{d(P_E+P_G)}dP_E \\ &= log2 + \int_\Omega log f_{EG} dP_E = log2 + \int_\Omega log f dP_E \end{aligned} \tag{3.8}$$

Where, $f = f_{EG}$. Then the integral term of the KL divergence expression given in Equation (3.8) over a region R (omiting the $\frac{1}{2}$ scale factor) is:

$$F(R) = \int_R log f dP_E \tag{3.9}$$

Next, we will show that F is equivalent to a reconstruction loss based on ℓ_0 norm. *i.e.* F is zero for any region in which $G(E(x)) \neq x$, and non-zero otherwise.

Proposition 1: $f > 0$ always holds.

Proof. Let $R^{f=0} = \{(x,z) \in \Omega : f(x,z) = 0\}$ be the region of Ω in which $f = 0$. Then the measure $p_E(R^{f=0}) = \int_{R^{f=0}} f d(P_E + P_G) = 0$. Hence $f > 0$ always holds. **Proposition 1** ensures Equation (3.9) is always well-defined. □

Proposition 2: F outside the support of P_G is zero. *i.e.* $F(\Omega \setminus supp(P_G)) = 0$

Proof. We'll show in region $R_S = \Omega \setminus supp(P_G)$, $f = 1$ always holds. Let $R^{f<1} = \{(x,z) \in R_s : f(x,z) < 1$ be the region of R_s in which $f < 1$. Then $P_E(R^{f<1}) = \int_{R^{f<1}} f d(P_E+P_G) = \int_{R^{f<1}} \underbrace{f}_{\leqslant \varepsilon \leqslant 1} dP_E + \underbrace{\int_{R^{f<1}} f dP_G}_{=0} \leqslant \varepsilon P_E(R^{f<1}) < P_E(R^{f<1})$. where ε is a constant smaller than 1. But $P_E(R^{f<1}) < P_E(R^{f<1})$ is a contradiction. Hence, $log f = 0$ in R_s, then we have F outside the support of P_G is zero. Notice that, by definition, $F(\Omega \setminus supp(P_E)) = 0$ always holds. So, only in region $R^1 = supp(P_E) \cap supp(P_G)$, F can be non-zero. □

Proposition 3: $f < 1$ holds in R^1

Proof. Let $R^{f=1} = \{(x,z) \in R^1 : f(x,z) = 1\}$ be the region in which $f = 1$. Let's assume that the set $R^{f=1} \neq \varnothing$ is not empty. By definition of support, we have $P_E(R^{f=1}) > 0$, $P_G(R^{f=1}) > 0$. On the other hand, the Radon Nikodym derivative on $R^{f=1}$ can be computed as: $P_E(R^{f=1}) = \int_{R^{f=1}} f d(P_E + P_G) = \int_{R^{f=1}} 1 d(P_E + P_G) = P_E(R^{f=1}) + P_G(R^{f=1})$, which implies $P_G(R^{f=1}) = 0$, and contradicts with the definition of support. Hence, $f < 1$ holds in R^1. □

Next, we will prove SelfExGAN introduce a ℓ_0 based reconstruction: **Theorem 2:** Equation (3.3) mimic a ℓ_0 based autoencoder.

$$\begin{aligned} &V(G,E,D^*_{EG}) \\ &= \mathbb{E}_{x\sim p_X}[\mathbf{1}_{[E(x)\in\Omega^s_Z \wedge G(E(x))=x]} log f(x,E(x))] + \\ &\quad \mathbb{E}_{z\sim p_Z}[\mathbf{1}_{[G(z)\in\Omega^s_X \wedge E(G(z))=z]} log(1 - f(G(z),z))] \end{aligned}$$

Proof. **Proposition 2** implies that R^1 is the only region of Ω where F may be non-zero. If we denote that:

$$supp(P_E) = \{(x, E(x)) : x \in \Omega_X^s\}$$
$$supp(P_G) = \{(G(z), z) : z \in \Omega_Z^s\}$$

where $supp(P_X) = \Omega_X^s$, $supp(P_Z) = \Omega_Z^s$. Then $R^1 = supp(P_E) \cap supp(P_G) = \{(x, z) : x \in \Omega_X^s \wedge z \in \Omega_Z^s \wedge E(x) = z \wedge G(z) = x\}$. Then the KL divergence of Equation (3.8) can be rewritten as (notice we can omit the $x \in \Omega_X^s$ condition from inside an expectation over P_X):

$$\begin{aligned}
&D_{KL}(P_E||\frac{P_E + P_G}{2}) - log2 \\
&= F(\Omega) = F(R^1) \\
&= \int_{R^1} logf(x, z)dP_E \\
&= \int_{\Omega} \mathbf{1}_{[(x,z)\in R^1]} logf(x, z)dP_E \\
&= \mathbb{E}_{(x,z)\sim p_E}[\mathbf{1}_{[(x,z)\in R^1]} logf(x, z)] \\
&= \mathbb{E}_{x\sim p_X}[\mathbf{1}_{[(x,E(x))\in R^1]} logf(x, E(x))] \\
&= \mathbb{E}_{x\sim p_X}[\mathbf{1}_{[E(x)\in\Omega_Z^s \wedge G(E(x))=x]} logf(x, E(x))]
\end{aligned} \tag{3.10}$$

Along with the same flow for deduction Equation (3.10), we can have:

$$\begin{aligned}
&D_{KL}(P_G||\frac{P_E + P_G}{2}) - log2 \\
&= \mathbb{E}_{z\sim p_Z}[\mathbf{1}_{[G(z)\in\Omega_X^s \wedge E(G(z))=z]} logf_{GE}(G(z), z)] \\
&= \mathbb{E}_{z\sim p_Z}[\mathbf{1}_{[G(z)\in\Omega_X^s \wedge E(G(z))=z]} log(1 - f(G(z), z))]
\end{aligned} \tag{3.11}$$

Then given the optimal D_{EG}^*, and substitute Equation (3.10),(3.11) into Equation (3.4), we have the ℓ_0 based reconstruction value function as:

$$\begin{aligned}
&V(G, E, D_{EG}^*) = 2D_{JS}(P_E||P_G) - 2log2 \\
&= \mathbb{E}_{x\sim p_X}[\mathbf{1}_{[E(x)\in\Omega_Z^s \wedge G(E(x))=x]} logf(x, E(x))] + \\
&\quad \mathbb{E}_{z\sim p_Z}[\mathbf{1}_{[G(z)\in\Omega_X^s \wedge E(G(z))=z]} log(1 - f(G(z), z))]
\end{aligned}$$

□

Our experiments show excellent results on reconstruction for the input data.

Chapter 4

Experiments and Applications

In this chapter, we first present our implementation details with our model which includes the model architectures and hyper-parameter settings. Then we give some discussion on our experimental results. In Section 4.2 we show our theory can have various applications.

4.1 Experiments

In this section, we give our implementation details with our model on different datasets: MNIST [2], SVHN [63], Cifar10 [64], Imagenet [3], CelebA [9], and cartoon face datasets.

4.1.1 Implementation Details

Our model consists of four sub-nets which are generator (G), encoder (E), and two different discriminator $D1$ and $D2$. The input in G net is a vector, given different datasets, we use the different length of vecotr. Generally speaking, a more complicated dataset will use the longer vector. The output in G net is generated image. We basically adapt the architecture of the DCGAN [8] implementation *i.e.* use deconvolutional layer for translating a vector into image. The architecture of E is similar to Alexnet [3]. The input in E is the image with size 32×32 or 64×64 depend on the dataset. The output of E is a vector sharing the same length with the input in G. Discriminator $D1$ tries to distinguish the real image with generated fake one, as we introduced in Chapter 3, the input in D is the image, the output is the probability of this input is whether a real one. $D1$ also uses an architecture in DCGAN [8]. The objective of discriminator $D2$ is distinguishing real pair with a fake pair, its input is the image with the latent vector. Similar to $D1$, the output is the probability of determining whether the input is a real pair. We use a MLP like [21] architecture to manage this job. In Section 4.1.2, we exhibit the proposed model architectures for different datasets. As we mentioned in Chapter 3, we do **not** use any extra tricks in architectures design as well as in training procedure. We use Adam solver [39] with $\beta_1 = 0.9$, $\beta_2 = 0.99$, learning rate: 0.001 for all the four sub-nets. We choose Adam because of its stable gradient backpropagation.

Another implementation detail we would like to mention is the generating for the latent vector. In normal GAN, the latent vector will be sampled from a random distribution, for example, Gaussian distribution. In our case, we learn the latent vector introduced by E along with sampling from a Gaussian distribution. When the latent vector is sampled from Gaussian distribution, $D2$ will not work (cause there exist no pair at all), the gradient computed from discriminator side will only by $D1$. The objective of SelfExGAN is to determine the correspondence between latent space and data space. When adapting the latent vector which is computed by E, we are setting an anchor point on latent space make

it corresponds to a point in data space. When sampling latent vector from a Gaussian distribution, we are setting the point around our anchor points. For preventing E project the data concentratedly to some specific region on the latent space, we start by projecting the latent vector into a sphere:

1. Step 1: Write the point in a coordinate system centered at the center of sphere (x_0, y_0, z_0):

$$P = (x', y', z') = (x - x_0, y - y_0, z - z_0) \tag{4.1}$$

2. Step 2: Compute the length of the vector:

$$|P| = sqrt(x'^2 + y'^2 + z'^2) \tag{4.2}$$

3. Step 3: Scale the vector so that it has length equal to the radius of the sphere:

$$Q = (radius/|P|) * P \tag{4.3}$$

4. Step 4: Change back to your original coordinate system to get the projection:

$$R = Q + (x_0, y_0, z_0) \tag{4.4}$$

then apply KL divergence from output of E to a unit Gaussian distribution as below:

$$KL(E(x)||N(0,1) = -\frac{M}{2} + \frac{1}{M}\sum_{i=1}^{M}\frac{{\sigma_i}^2 + {\mu_i}^2}{2} - log(\sigma_i) \tag{4.5}$$

M is length of the latent vector, μ_i is the mean of ith dimension on the latent vector, σ_i is the variance of the latent vector. For the derivation of Equation (4.5), please refer to *Appendix A*

4.1.2 Model Architectures

In this section, we give the tables summarize our net architectures for different datasets as well as the results.

Figure 4.1 shows our results on MNIST dataset. Figure 4.1a is the real samples of MNIST dataset. Figure 4.1b shows the reconstruction one on Figure 4.1a. *i.e.*, we put real images into encoder, then use the computed latent vector to generate images. Figure 4.1 shows when sampling the latent vector on a unit Gaussian distribution. Tabel 4.1 summarizes the architecture for MNIST dataset. The image size of original MNIST is 28×28, in our implementation, we resize it to 32×32. The length of latent vector is 64. We use 30000 images for training.

Figure 4.2 shows our results on SVHN dataset. Figure 4.2a is the real samples of SVHN dataset. Figure 4.2b shows the reconstruction one on Figure 4.2a. *i.e.*, we put real images into encoder, then use the computed latent vector to generate images. Figure 4.2c shows when sampling the latent vector on a unit Gaussian distribution. Tabel 4.2 summarizes the architecture for SVHN dataset. Input size as for MNIST dataset, 32×32, we use 40000 images for training, latent vector length as 128.

Figure 4.3 shows our results on Cifar10 dataset. Figure 4.3a is the real samples of Cifar10 dataset. Figure 4.3b shows the reconstruction one on Figure 4.3a. *i.e.*, we put real images into encoder, then use the computed latent vector to generate images. Figure 4.3c shows when sampling the latent vector on a unit Gaussian distribution. Tabel 4.3

(a) Samples on MNIST dataset.

(b) Reconstructions on MNIST dataset.

(c) Generated images on MNIST data.

Fig. 4.1: Results on MNIST dataset.

Table 4.1: MNIST model parameter setting.

Operations	Kernel	Strides	Feature map	Use BN ?	Non-linearity
G(z) input: z					
(z) - $1 \times 1 \times 64$:					
Fully-connect			$1 \times 1 \times 4096$	yes	ReLU
Reshape			$4 \times 4 \times 256$		
Deconv	5×5	2	$8 \times 8 \times 128$	yes	ReLU
Deconv	5×5	2	$16 \times 16 \times 64$	yes	ReLU
Deconv	5×5	2	$32 \times 32 \times 1$	no	Tanh
E(x) input: x					
(x) - $32 \times 32 \times 1$:					
Conv	3×3	1	$32 \times 32 \times 64$	yes	ReLu
Max-pool	2×2		16×16		
Conv	3×3	1	$16 \times 16 \times 128$	yes	ReLu
Max-pool	2×2		8×8		
Conv	3×3	1	$8 \times 8 \times 256$	yes	ReLu
Max-pool	2×2		4×4		
Flatten					
Fully-connect			$1 \times 1 \times 64$	no	Sigmoid
D1(x) input: x					
(x) - $32 \times 32 \times 1$:					
Conv	5×5	2	$16 \times 16 \times 64$	yes	ReLu
Conv	5×5	2	$8 \times 8 \times 128$	yes	ReLu
Conv	5×5	2	$4 \times 4 \times 256$	yes	ReLu
Conv	5×5	2	2×2	yes	ReLu
Flatten					
Fully-connect			$1 \times 1 \times 1$	no	Sigmoid
D2(x,z) inputs: [x ,z]					
(x) - $32 \times 32 \times 1$:					
Flatten					
Fully-connect			$1 \times 1 \times 32 * 32$	yes	ReLu
Fully-connect			$1 \times 1 \times 256$	yes	ReLu
Fully-connect			$1 \times 1 \times 64$	no	ReLu
(z) - $1 \times 1 \times 64$:					
Concat			$1 \times 1 \times 128$		
Fully-connect			$1 \times 1 \times 256$	no	ReLu
Fully-connect			$1 \times 1 \times 128$	yes	ReLu
Fully-connect			$1 \times 1 \times 64$	yes	ReLu
Fully-connect			$1 \times 1 \times 1$	no	Sigmoid

(a) Samples on SVHN dataset.

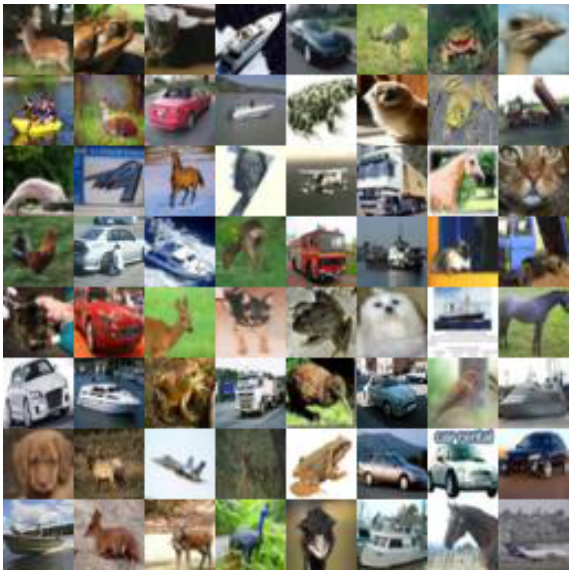
(b) Reconstructions on SVHN dataset.

(c) Generated images on SVHN data.

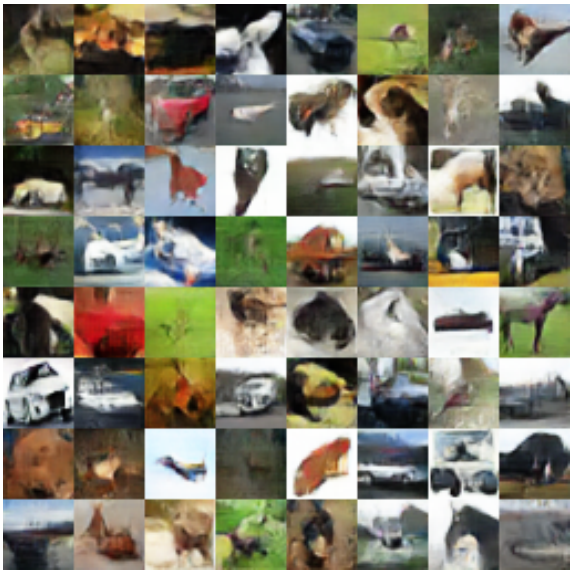
Fig. 4.2: Results on SVHN dataset.

Table 4.2: SVHN model parameter setting.

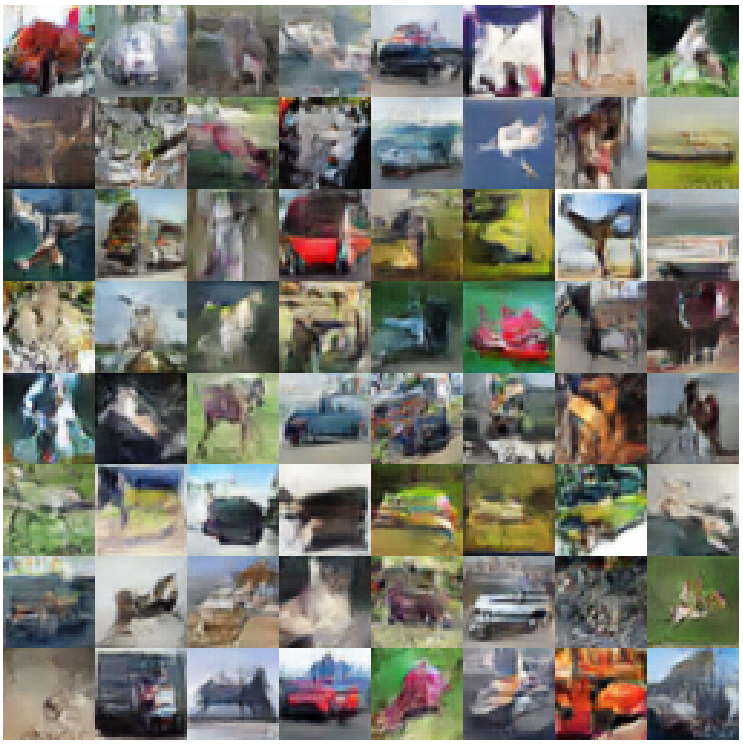
Operations	Kernel	Strides	Feature map	Use BN ?	Non-linearity
G(z) input: z					
(z) - $1 \times 1 \times 128$:					
Fully-connect			$1 \times 1 \times 4096$	yes	ReLU
Reshape			$4 \times 4 \times 256$		
Deconv	5×5	2	$8 \times 8 \times 128$	yes	ReLU
Deconv	5×5	2	$16 \times 16 \times 64$	yes	ReLU
Deconv	5×5	2	$32 \times 32 \times 3$	no	Tanh
E(x) input: x					
(x) - $32 \times 32 \times 3$:					
Conv	3×3	1	$32 \times 32 \times 64$	yes	ReLu
Max-pool	2×2		16×16		
Conv	3×3	1	$16 \times 16 \times 128$	yes	ReLu
Max-pool	2×2		8×8		
Conv	3×3	1	$8 \times 8 \times 256$	yes	ReLu
Max-pool	2×2		4×4		
Flatten					
Fully-connect			$1 \times 1 \times 128$	no	Sigmoid
D1(x) input: x					
(x) - $32 \times 32 \times 3$:					
Conv	5×5	2	$16 \times 16 \times 64$	yes	ReLu
Conv	5×5	2	$8 \times 8 \times 128$	yes	ReLu
Conv	5×5	2	$4 \times 4 \times 256$	yes	ReLu
Conv	5×5	2	2×2	yes	ReLu
Flatten					
Fully-connect			$1 \times 1 \times 2$	no	Softmax
D2(x,z) inputs: [x ,z]					
(x) - $32 \times 32 \times 3$:					
Flatten					
Fully-connect			$1 \times 1 \times 32 * 32$	yes	ReLu
Fully-connect			$1 \times 1 \times 256$	yes	ReLu
Fully-connect			$1 \times 1 \times 128$	no	ReLu
(z) - $1 \times 1 \times 128$:					
Concat			$1 \times 1 \times 256$		
Fully-connect			$1 \times 1 \times 512$	no	ReLu
Fully-connect			$1 \times 1 \times 256$	yes	ReLu
Fully-connect			$1 \times 1 \times 128$	yes	ReLu
Fully-connect			$1 \times 1 \times 64$	yes	ReLu
Fully-connect			$1 \times 1 \times 2$	no	Softmax



(a) Samples on cifar10 dataset.



(b) Reconstructions Cifar10 dataset.



(c) Generated images on cifar10 dataset.

Fig. 4.3: Results on Cifar10 dataset.

Table 4.3: Cifar10 model parameter setting.

Operations	Kernel	Strides	Feature map	Use BN ?	Non-linearity
G(z) input: z					
(z) - $1 \times 1 \times 128$:					
Fully-connect			$1 \times 1 \times 4096$	yes	ReLU
Reshape			$4 \times 4 \times 256$		
Deconv	5×5	2	$8 \times 8 \times 128$	yes	ReLU
Deconv	5×5	2	$16 \times 16 \times 64$	yes	ReLU
Deconv	5×5	2	$32 \times 32 \times 3$	no	Tanh
E(x) input: x					
(x) - $32 \times 32 \times 3$:					
Conv	5×5	1	$32 \times 32 \times 64$	yes	LeakyReLu
Max-pool	2×2		16×16		
Conv	5×5	1	$16 \times 16 \times 128$	yes	LeakyReLu
Max-pool	2×2		8×8		
Conv	5×5	1	$8 \times 8 \times 256$	yes	LeakyReLu
Max-pool	2×2		4×4		
Flatten					
Fully-connect			$1 \times 1 \times 128$	no	Sigmoid
D1(x) input: x					
(x) - $32 \times 32 \times 3$:					
Conv	5×5	2	$16 \times 16 \times 64$	yes	LeakyReLu
Conv	5×5	2	$8 \times 8 \times 128$	yes	LeakyReLu
Conv	5×5	2	$4 \times 4 \times 256$	yes	LeakyReLu
Conv	5×5	2	2×2	yes	LeakyReLu
Flatten					
Fully-connect			$1 \times 1 \times 2$	no	Softmax
D2(x,z) inputs: [x ,z]					
(x) - $32 \times 32 \times 3$:					
Flatten					
Fully-connect			$1 \times 1 \times 32 * 32$	yes	ReLu
Fully-connect			$1 \times 1 \times 256$	yes	ReLu
Fully-connect			$1 \times 1 \times 128$	no	ReLu
(z) - $1 \times 1 \times 128$:					
Concat			$1 \times 1 \times 256$		
Fully-connect			$1 \times 1 \times 256$	no	ReLu
Dropout (0.1)					
Fully-connect			$1 \times 1 \times 512$	yes	ReLu
Dropout (0.1)					
Fully-connect			$1 \times 1 \times 256$	yes	ReLu
Fully-connect			$1 \times 1 \times 128$	yes	ReLu
Fully-connect			$1 \times 1 \times 64$	yes	ReLu
Fully-connect			$1 \times 1 \times 2$	no	Softmax

summarizes the architecture for Cifar10 dataset. The input size 32×32, we use 35000 images for training, latent vector length as 128.

Figure 4.4 shows our results on Imagenet dataset. Figure 4.4a is the real samples of Imagenet dataset. Figure 4.4b shows the reconstruction one on Figure 4.4a. *i.e.*, we put real images into encoder, then use the computed latent vector to generate images. Figure 4.4c shows when sampling the latent vector on a unit Gaussian distribution. Tabel 4.4 summarizes the architecture for Imagenet dataset. The input size 32×32, we use 40000 images for training, latent vector length as 128.

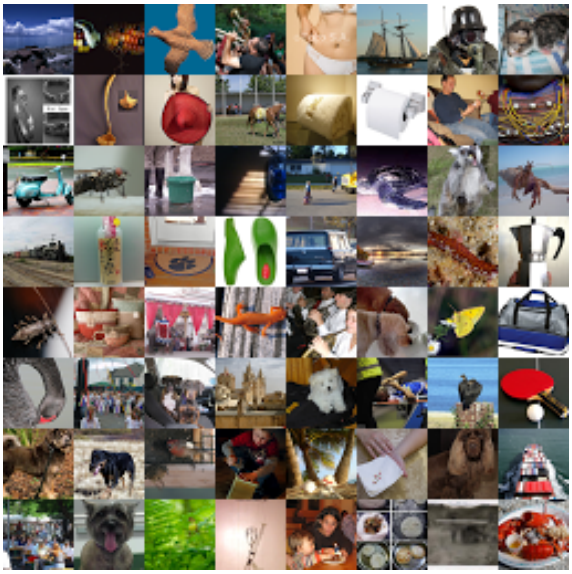
For better observing the facial detail, we also do experiment on a center cropped face dataset (64×64 center crop on CelebA dataset). Figure 4.5 shows our results. Figure 4.5a is the real samples. Figure 4.5b shows the reconstruction one on Figure 4.4a. *i.e.*, we put real images into encoder, then use the computed latent vector to generate images. Figure 4.5c shows when sampling the latent vector on a unit Gaussian distribution. Tabel 4.5 summarizes the architecture for cropped face dataset. The input size 64×64, we use 40000 images for training, latent vector length as 64.

Figure 4.6 shows our results on CelebA dataset. Figure 4.6a is the real samples of CelebA dataset. Figure 4.6b shows the reconstruction one on Figure 4.6a. *i.e.*, we put real images into encoder, then use the computed latent vector to generate images. Figure 4.6c shows when sampling the latent vector on a unit Gaussian distribution. Tabel 4.6 summarizes the architecture for CelebA dataset. The input size 64×64, we use 40000 images for training, latent vector length as 128.

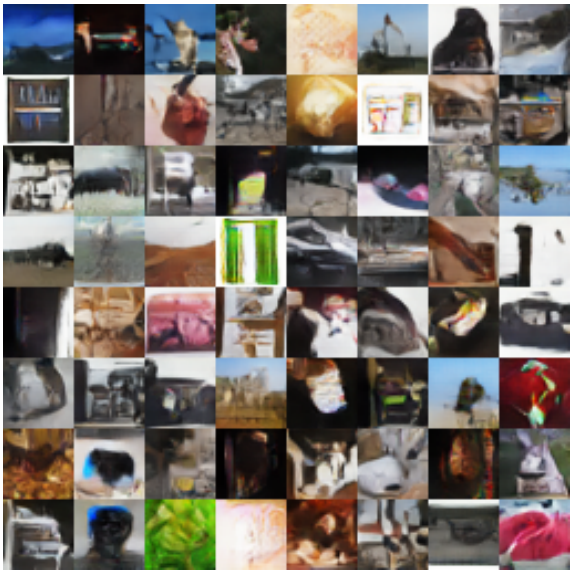
For better show the learning ability of our proposed model. We crawl a dataset of cartoon dataset. Figure 4.7 shows our results on the collected cartoon dataset. Figure 4.7a is the real samples of cartoon dataset. Figure 4.7b shows the reconstruction one on Figure 4.7a. *i.e.*, we put real images into encoder, then use the computed latent vector to generate images. Figure 4.7c shows when sampling the latent vector on a unit Gaussian distribution. Tabel 4.7 summarizes the architecture for cartoon dataset. The input size 64×64, we use 40000 images for training, latent vector length as 128.

4.1.3 Discussion

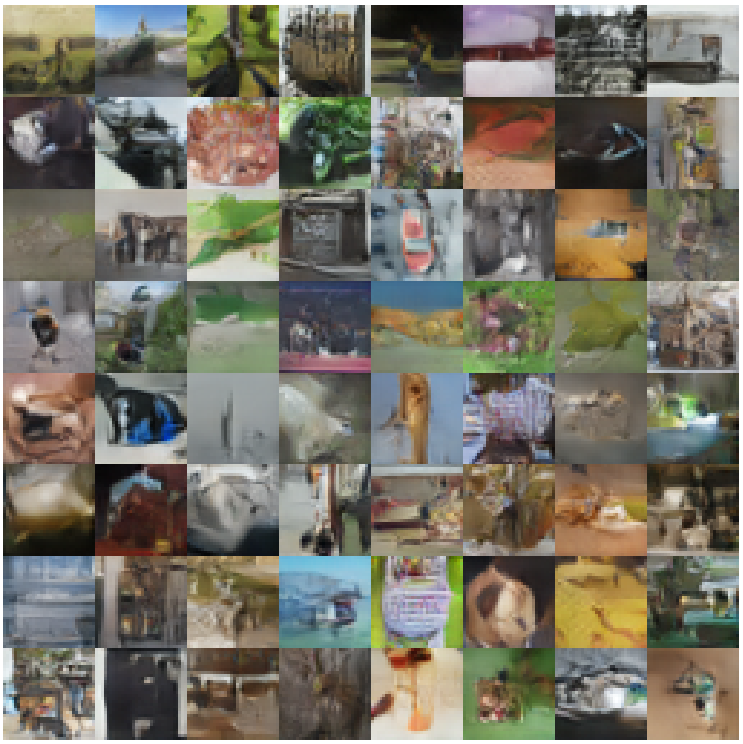
Figure 4.8 shows that given a testing image (feed it into E and input the corresponding z into G) when iteratively keep feeding with the former generated sample, the generated image will gradually change from one number to another number (Figure 4.8a), one face to another (Figure 4.8b). We observe the similar phenomenon on the other datasets. This demonstrates that the proposed SelfExGAN is able to discover the hidden relationship lying behind the training data. Moreover, SelfExGAN can determine the correspondence between the latent input and its memorized training data. Along with the latent input changing, the generated image will change accordingly. Given a latent input, SelfExGAN can reckon by itself and generate a corresponding sample but not a likeness of any specific training data. Otherwise, SelfExGAN cannot keep gradually generating different faces iteratively. The interesting thing is when the input sample has a unique pattern that does not share any feature with other samples, SelfExGAN can not generating from one sample to another. For example, it fails to generate number 1 to another sample (Figure 4.8a), or generate from a man with sunglass cover most part of his face to another face (Figure 4.8b). This act makes sense because when projecting a unique data into latent space, in latent space, it will still keep its uniqueness.



(a) Samples on Imagenet dataset.



(b) Reconstructions on Imagenet dataset.



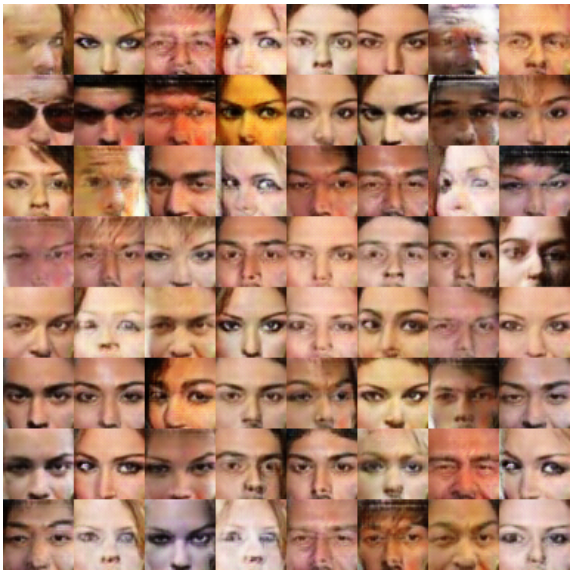
(c) Generated images on Imagenet dataset.

Fig. 4.4: Results on Imagenet dataset.

Table 4.4: Imagenet model parameter setting.

Operations	Kernel	Strides	Feature map	Use BN ?	Non-linearity
G(z) input: z					
(z) - $1 \times 1 \times 128$:					
Fully-connect			$1 \times 1 \times 4096$	yes	ReLU
Reshape			$4 \times 4 \times 256$		
Deconv	5×5	2	$8 \times 8 \times 128$	yes	ReLU
Deconv	5×5	2	$16 \times 16 \times 64$	yes	ReLU
Deconv	5×5	2	$32 \times 32 \times 3$	no	Tanh
E(x) input: x					
(x) - $32 \times 32 \times 3$:					
Conv	5×5	1	$32 \times 32 \times 64$	yes	LeakyReLu
Max-pool	2×2		16×16		
Conv	5×5	1	$16 \times 16 \times 128$	yes	LeakyReLu
Max-pool	2×2		8×8		
Conv	5×5	1	$8 \times 8 \times 256$	yes	LeakyReLu
Max-pool	2×2		4×4		
Flatten					
Fully-connect			$1 \times 1 \times 128$	no	Sigmoid
D1(x) input: x					
(x) - $32 \times 32 \times 3$:					
Conv	5×5	2	$16 \times 16 \times 64$	yes	LeakyReLu
Conv	5×5	2	$8 \times 8 \times 128$	yes	LeakyReLu
Conv	5×5	2	$4 \times 4 \times 256$	yes	LeakyReLu
Conv	5×5	2	2×2	yes	LeakyReLu
Flatten					
Fully-connect			$1 \times 1 \times 2$	no	Softmax
D2(x,z) inputs: [x ,z]					
(x) - $32 \times 32 \times 3$:					
Flatten					
Fully-connect			$1 \times 1 \times 32 * 32$	yes	ReLu
Fully-connect			$1 \times 1 \times 256$	yes	ReLu
Fully-connect			$1 \times 1 \times 128$	no	ReLu
(z) - $1 \times 1 \times 128$:					
Concat			$1 \times 1 \times 256$		
Fully-connect			$1 \times 1 \times 256$	no	ReLu
Fully-connect			$1 \times 1 \times 512$	yes	ReLu
Dropout (0.1)					
Fully-connect			$1 \times 1 \times 1024$	yes	ReLu
Dropout (0.1)					
Fully-connect			$1 \times 1 \times 512$	yes	ReLu
Dropout (0.1)					
Fully-connect			$1 \times 1 \times 256$	yes	ReLu
Fully-connect			$1 \times 1 \times 128$	yes	ReLu
Fully-connect			$1 \times 1 \times 64$	yes	ReLu
Fully-connect			$1 \times 1 \times 2$	no	Softmax

(a) Samples on CelebA dataset (with cropped).



(b) Reconstructions reuslts.

(c) Generated images on CelebA dataset with cropped.

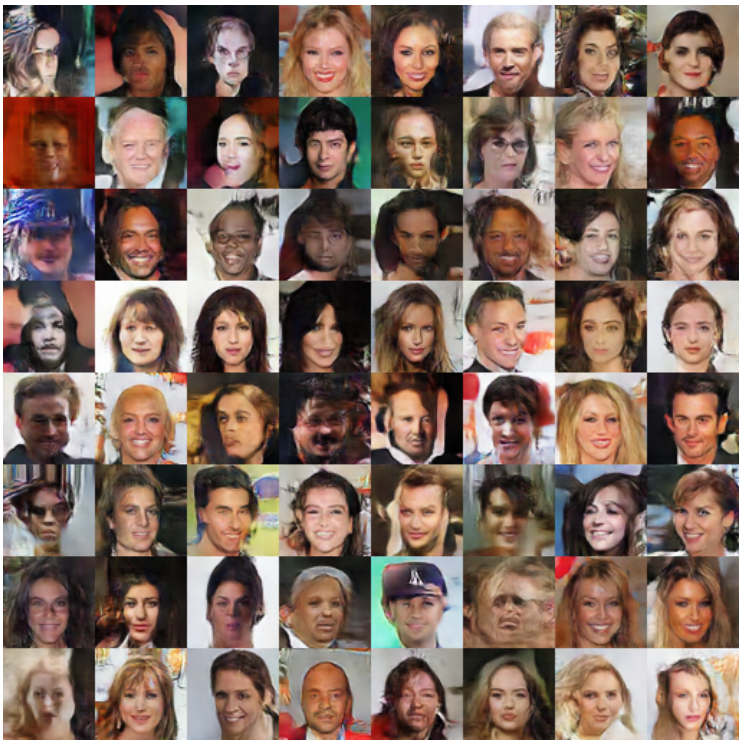
Fig. 4.5: Results on CelebA crop dataset.

Table 4.5: CelebA(with center crop) model parameter setting.

Operations	Kernel	Strides	Feature map	Use BN ?	Non-linearity
G(z) input: z					
(z) - $1 \times 1 \times 64$:					
Fully-connect			$1 \times 1 \times 4096$	yes	ReLU
Reshape			$4 \times 4 \times 256$		
Deconv	3×3	2	$8 \times 8 \times 128$	yes	LeakyReLU
Deconv	3×3	2	$16 \times 16 \times 64$	yes	LeakyReLU
Deconv	3×3	2	$32 \times 32 \times 64$	yes	LeakyReLU
Deconv	5×5	2	$32 \times 32 \times 3$	no	Tanh
E(x) input: x					
(x) - $64 \times 64 \times 3$:					
Conv	3×3	1	$64 \times 64 \times 64$	yes	ReLu
Max-pool	2×2		32×32		
Conv	3×3	1	$32 \times 32 \times 64$	yes	ReLu
Max-pool	2×2		16×16		
Conv	3×3	1	$16 \times 16 \times 128$	yes	ReLu
Max-pool	2×2		8×8		
Conv	3×3	1	$8 \times 8 \times 256$	yes	ReLu
Max-pool	2×2		4×4		
Flatten					
Fully-connect			$1 \times 1 \times 64$	no	Sigmoid
D1(x) input: x					
(x) - $64 \times 64 \times 3$:					
Conv	5×5	2	$32 \times 32 \times 64$	yes	LeakyReLu
Conv	5×5	2	$16 \times 16 \times 64$	yes	LeakyReLu
Conv	5×5	2	$8 \times 8 \times 128$	yes	LeakyReLu
Conv	5×5	2	$4 \times 4 \times 256$	yes	LeakyReLu
Conv	5×5	2	2×2	yes	LeakyReLu
Flatten					
Fully-connect			$1 \times 1 \times 2$	no	Softmax
D2(x,z) inputs: [x ,z]					
(x) - $64 \times 64 \times 3$:					
Conv	5×5	2	$32 \times 32 \times 64$	yes	LeakyReLu
Conv	5×5	2	$16 \times 16 \times 64$	yes	LeakyReLu
Conv	5×5	2	$8 \times 8 \times 128$	yes	LeakyReLu
Fully-connect			$1 \times 1 \times 64$	no	LeakyReLu
(z) - $1 \times 1 \times 64$:					
Concat			$1 \times 1 \times 128$		
Fully-connect			$1 \times 1 \times 256$	no	ReLu
Fully-connect			$1 \times 1 \times 128$	yes	ReLu
Fully-connect			$1 \times 1 \times 64$	yes	ReLu
Fully-connect			$1 \times 1 \times 2$	no	Softmax

(a) Samples on CelebA dataset.

(b) Reconstructions on CelebA dataset.



(c) Generated images on CelebA dataset.

Fig. 4.6: Results on CelebA dataset.

Table 4.6: CelebA(with resized to 64×64) model parameter setting.

Operations	Kernel	Strides	Feature map	Use BN ?	Non-linearity
G(z) input: z					
(z) - $1 \times 1 \times 128$:					
Fully-connect			$1 \times 1 \times 4096$	yes	ReLU
Reshape			$4 \times 4 \times 256$		
Deconv	3×3	2	$8 \times 8 \times 128$	yes	LeakyReLU
Deconv	3×3	2	$16 \times 16 \times 64$	yes	LeakyReLU
Deconv	3×3	2	$32 \times 32 \times 64$	yes	LeakyReLU
Deconv	5×5	2	$32 \times 32 \times 3$	no	Tanh
E(x) input: x					
(x) - $64 \times 64 \times 3$:					
Conv	3×3	1	$64 \times 64 \times 64$	yes	ReLu
Max-pool	2×2		32×32		
Conv	3×3	1	$32 \times 32 \times 64$	yes	ReLu
Max-pool	2×2		16×16		
Conv	3×3	1	$16 \times 16 \times 128$	yes	ReLu
Max-pool	2×2		8×8		
Conv	3×3	1	$8 \times 8 \times 256$	yes	ReLu
Max-pool	2×2		4×4		
Flatten					
Fully-connect			$1 \times 1 \times 128$	no	Sigmoid
D1(x) input: x					
(x) - $64 \times 64 \times 3$:					
Conv	5×5	2	$32 \times 32 \times 64$	yes	LeakyReLu
Conv	5×5	2	$16 \times 16 \times 64$	yes	LeakyReLu
Conv	5×5	2	$8 \times 8 \times 128$	yes	LeakyReLu
Conv	5×5	2	$4 \times 4 \times 256$	yes	LeakyReLu
Conv	5×5	2	2×2	yes	LeakyReLu
Flatten					
Fully-connect			$1 \times 1 \times 2$	no	Softmax
D2(x,z) inputs: [x ,z]					
(x) - $64 \times 64 \times 3$:					
Conv	5×5	2	$32 \times 32 \times 64$	yes	LeakyReLu
Conv	5×5	2	$16 \times 16 \times 64$	yes	LeakyReLu
Conv	5×5	2	$8 \times 8 \times 128$	yes	LeakyReLu
Fully-connect			$1 \times 1 \times 64$	no	LeakyReLu
(z) - $1 \times 1 \times 128$:					
Concat			$1 \times 1 \times 256$		
Fully-connect			$1 \times 1 \times 256$	no	ReLu
Fully-connect			$1 \times 1 \times 128$	yes	ReLu
Fully-connect			$1 \times 1 \times 64$	yes	ReLu
Fully-connect			$1 \times 1 \times 2$	no	Softmax

(a) Samples on cartoon dataset.

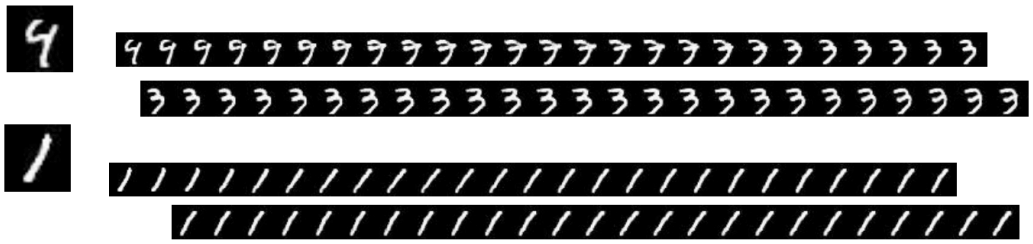
(b) Reconstrations on cartoon dataset.

(c) Generated images on cartoon dataset.

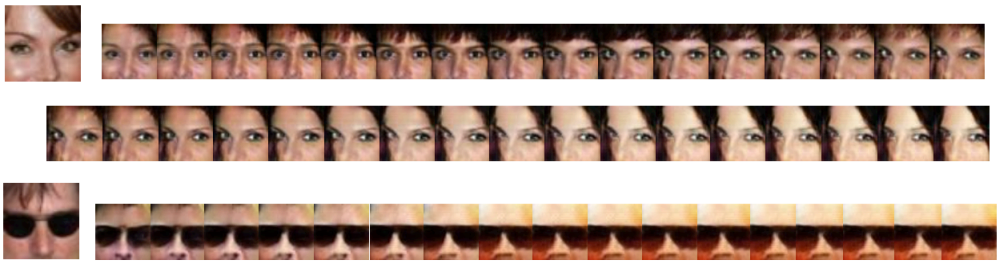
Fig. 4.7: Results on cartoon dataset.

Table 4.7: Cartoon model parameter setting.

Operations	Kernel	Strides	Feature map	Use BN ?	Non-linearity
G(z) input: z					
(z) - $1 \times 1 \times 128$:					
Fully-connect			$1 \times 1 \times 4096$	yes	ReLU
Reshape			$4 \times 4 \times 256$		
Deconv	5×5	2	$8 \times 8 \times 128$	yes	LeakyReLU
Deconv	5×5	2	$16 \times 16 \times 64$	yes	LeakyReLU
Deconv	5×5	2	$32 \times 32 \times 64$	yes	LeakyReLU
Deconv	5×5	2	$32 \times 32 \times 3$	no	Tanh
E(x) input: x					
(x) - $64 \times 64 \times 3$:					
Conv	5×5	1	$64 \times 64 \times 64$	yes	ReLu
Max-pool	2×2		32×32		
Conv	5×5	1	$32 \times 32 \times 64$	yes	ReLu
Max-pool	2×2		16×16		
Conv	5×5	1	$16 \times 16 \times 128$	yes	ReLu
Max-pool	2×2		8×8		
Conv	3×3	1	$8 \times 8 \times 256$	yes	ReLu
Max-pool	2×2		4×4		
Flatten					
Fully-connect			$1 \times 1 \times 128$	no	Sigmoid
D1(x) input: x					
(x) - $64 \times 64 \times 3$:					
Conv	5×5	2	$32 \times 32 \times 64$	yes	LeakyReLu
Conv	5×5	2	$16 \times 16 \times 64$	yes	LeakyReLu
Conv	5×5	2	$8 \times 8 \times 128$	yes	LeakyReLu
Conv	5×5	2	$4 \times 4 \times 256$	yes	LeakyReLu
Conv	5×5	2	2×2	yes	LeakyReLu
Flatten					
Fully-connect			$1 \times 1 \times 2$	no	Softmax
D2(x,z) inputs: [x ,z]					
(x) - $64 \times 64 \times 3$:					
Conv	5×5	2	$32 \times 32 \times 64$	yes	LeakyReLu
Conv	5×5	2	$16 \times 16 \times 64$	yes	LeakyReLu
Conv	5×5	2	$8 \times 8 \times 128$	yes	LeakyReLu
Fully-connect			$1 \times 1 \times 64$	no	LeakyReLu
(z) - $1 \times 1 \times 128$:					
Concat			$1 \times 1 \times 256$		
Fully-connect			$1 \times 1 \times 256$	no	ReLu
Fully-connect			$1 \times 1 \times 128$	yes	ReLu
Fully-connect			$1 \times 1 \times 64$	yes	ReLu
Fully-connect			$1 \times 1 \times 2$	no	Softmax



(a) Iterative feeding on MNIST dataset.



(b) Iterative feeding on CelebA dataset with cropped.

Fig. 4.8: Given a real image, then iteratively keep feeding with the former generated sample.

4.2 Applications

In this section, we introduce three application with our proposed model and theory, which are generating training data with label for supervised learning, evaluation on the similarity of two samples, and improving the divergent creation for design.

4.2.1 Generating Training Data for Supervised Learning

Compare with unsupervised learning, supervised learning still takes a great portion in artificial intelligence applications. In this section, we show our proposed model can generate the training data for supervised learning.

For generating data with labels, let's consider in a classification mission. Our approach includes two steps. First, use the encoder E to project our target data in the latent space. Then with adding a noise on that encoded vector, we feed it with our generator for getting our synthesized data. Figure 4.9 shows a visualization of the 2-dimensional t-SNE [14] of the latent space representation on MNIST dataset. Figure 4.10 shows we can generate the fake data for training on MNIST, Cifar10, Imagenet datasets.

4.2.2 Similarity Evaluation

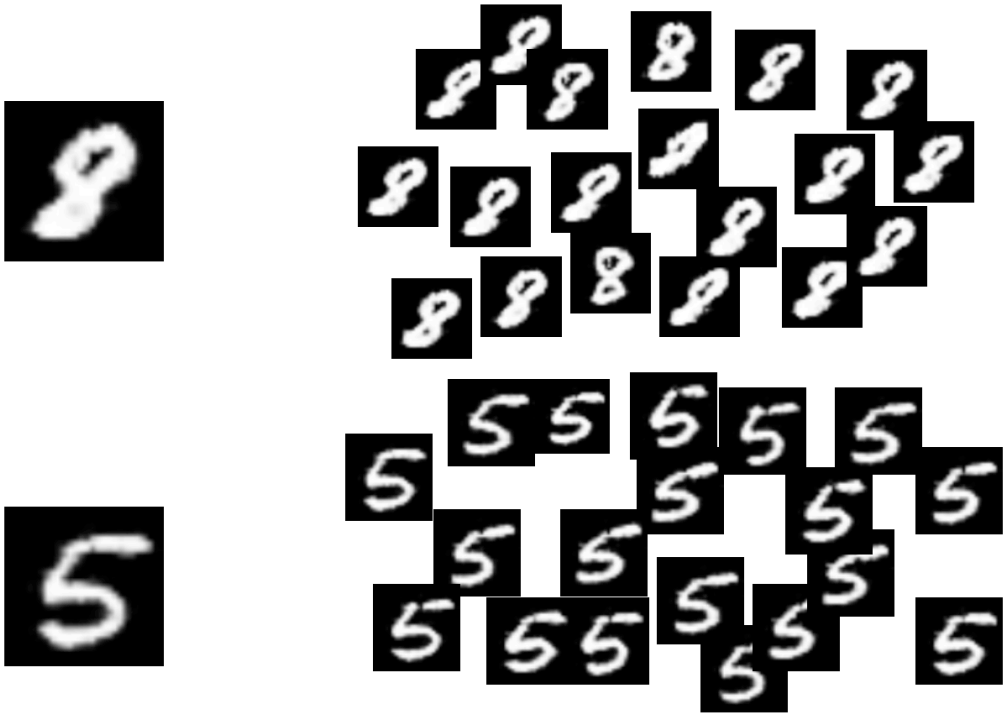
Our model can propose an application for similarity evaluation. For example, face comparing can be used in face-based user verification and personal identification. Figure 4.11 shows how we can check the likelihood of two faces by generating its "neighborhood"

Fig. 4.9: Visualization of the 2-dimensional t-SNE [14] of the latent space representation on MNIST dataset.

fake faces. It can be done by using the encoder to project the face into latent space, then use the generator to generate with the projected vector with noise adding. In Figure 4.11, we can understand the up two faces have lot of similarities, and the down face does look like to the up two.

4.2.3 Improving the Creation of Design

For a designer, a creative and open mind is very important. However, when a designer invents something, it usually happens that he or she will be restricted by his or her inertia thinking of the former design. Our proposed model introduces an application for improving the creation on the design shown in Figure 4.12. Say you design a cartoon face, and not sure if it is the best design. Then you can use our proposed model to manage an exploitation of your design. Figure 4.12 shows three type of results when adding with three different scales of noises.



(a) Generated data for MNIST dataset.

(b) Generated data for cifar10 dataset.

(c) Generated data on Imagenet dataset.

Fig. 4.10: Generate data for supervised learning.

Fig. 4.11: Similarity evaluation with human faces.

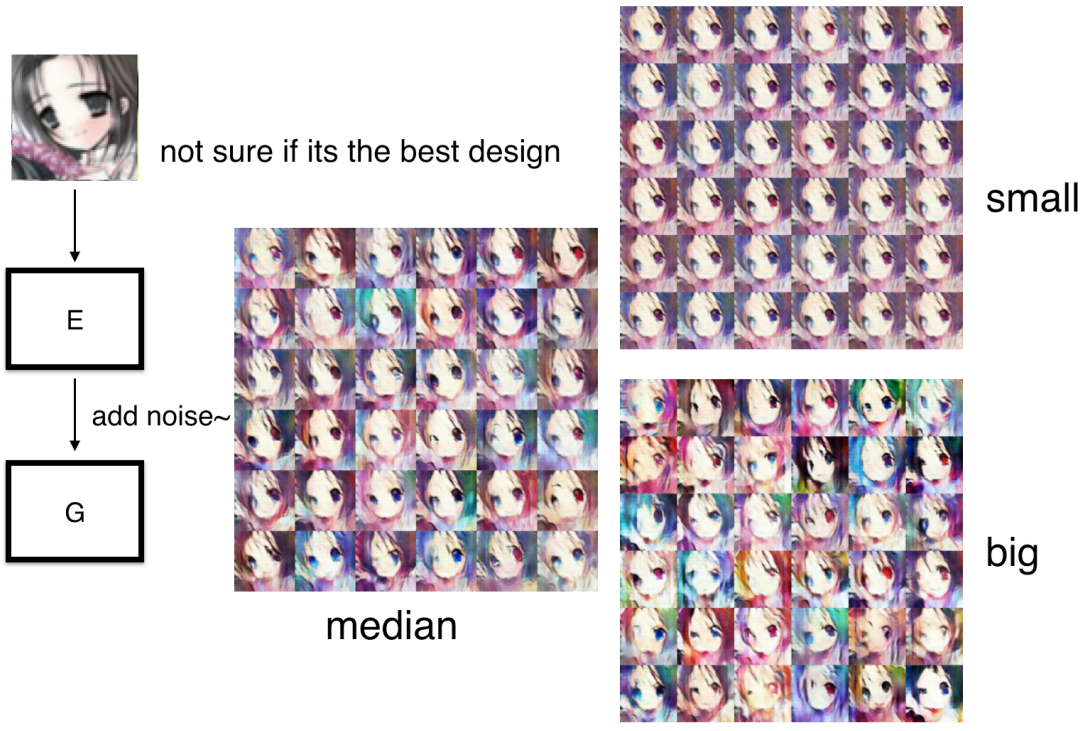


Fig. 4.12: Improving the divergent Creation.

Chapter 5

Conclusion and Future Works

In this thesis, we present a Generative Adversarial Network for latent space understanding. In Chapter 1, we give an introduction about deep learning along with some classic network architectures. In Chapter 2, we introduce the components in DNNs and GANs, and how to train a DNN by seeking a set of parameters for the network that minimize the objective function. Generative Adversarial Networks consists of two DNNs, which are called generator and discriminator separately. The main idea behind a GAN is to have these two competing neural network models trying to achieve opposite objectives. Generator takes random noise as input and tries to generate samples with high authenticity. On the other hand, discriminator receives samples from both the generator and the training data, and has to be able to distinguish between these two sources. G and D play a continuous game, where the generator is learning to produce more realistic samples, and the discriminator is learning to distinguish generated data from real one. These two networks are trained simultaneously, and the objective is that the competition will drive the generated samples to be indistinguishable from real ones. In Chapter 3, we introduce a latent space understandable GAN – SelfExGAN. SelfExGAN consists of three components, which are encoder (E), generator (G), and discriminator (D). Compared with other GAN based networks only take latent variable z as random input, SelfExGAN is trained to discover the "meaning" behind z. SelfExGAN is able to determine the correspondence between latent input and training data. SelfExGAN is trained in a self-excited way. That is, E takes the input from G's output, G takes the input sampled from E's output. E and G together constitute a positive feedback in SelfExGAN. The most attractive point of SelfExGAN is that it can use the learned correspondence to guide G to generate homomorphic samples given different latent inputs in an unsupervised learning manner. In Chapter 4, we first give our implementation details with our model which includes the model architectures and parameter settings. Then we give some discussion based our experiment results. Also, we show our theory can have various applications.

Note that we only do our experiment on small size of images with straightforward network architectures. In our understanding, we consider the neural network as the differentiable high-level function approximator. That means, for high-resolution image, we can just substitute straightforward network architectures with a more complicated network for the implementation of generator, encoder, and discriminators. However, our work does have some theoretical imitation. In our provided mathematical proof, we consider the two discriminators D1 and D2 as a holistic discriminator. This is **not** a strict proof. For the future works, we will fix our theory and expand our theory to other fields as well.

Thanks

This thesis is under the instruction of Professor Kamijo. He gives me the direction and advices not only on the researches but also on my life. Ms. Miwa and Ms. Hosaka, Professor Kamijo's secretaries, also provided a huge help to my research life in the lab. Also, researcher Yanlei Gu, Lei Wang, Qifeng Li and Shao Wang are the people who gives me a lot of important advice on my researches. Nothing in this thesis can be achieved without their help. To the other members in the lab, Jinwen Liu, Jiali Bao, Javanmardi Ehsan, Javanmardi Mahdi, Xu Liu, Haitao Wang, Lijia Xie, Bhattacharyya Prarthana, Qianlong Wang, Dailin Li, Ya Wang, Hirozuku Kitamaru, Yue Zhang, Huiyang Zhang, Youshihiko Kamiya, I want to thank them for companying me through the two years and make all these time more meaningful time. Lastly, I want to thank my family and friends for their supporting during this two years. In the end, we would like to thank the developers of tensorflow [31]. Thanks for their work, we can implement the deep neural network so easily.

January 31st, 2018

Publications

[1] Yongjie Liu, Qianlong Wang, Yanlei Gu, Shunsuke Kamijo “A Latent Space Understandable Generative Adversarial Network: SelfExGAN “ The International Conference on Digital Image Computing: Techniques and Applications (DICTA), Sydney, Australia, 29 Nov - 01 Dec 2017.

[2] Qianlong Wang, Yongjie Liu, Jingwen Liu, Yanlei Gu, Shunsuke Kamijo “Critical Areas Detection and Vehicle Speed Estimation System Towards Intersection-Related Driving Behavior Analysis” IEEE International Conference on Consumer Electronics, Las Vegas, American, January 12 - 14, 2018.

Reference

[1] R. Shapley and M. J. Hawken, "Color in the cortex: single-and double-opponent cells," *Vision research*, vol. 51, no. 7, pp. 701–717, 2011.

[2] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[4] M. D. Zeiler, D. Krishnan, G. W. Taylor, and R. Fergus, "Deconvolutional networks," in *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pp. 2528–2535, IEEE, 2010.

[5] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[6] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

[7] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[8] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *arXiv preprint arXiv:1511.06434*, 2015.

[9] Z. Liu, P. Luo, X. Wang, and X. Tang, "Deep learning face attributes in the wild," in *Proceedings of International Conference on Computer Vision (ICCV)*, 2015.

[10] T. White, "Sampling generative networks: Notes on a few effective techniques," *CoRR*, vol. abs/1609.04468, 2016.

[11] X. Chen, Y. Duan, R. Houthooft, J. Schulman, I. Sutskever, and P. Abbeel, "Infogan: Interpretable representation learning by information maximizing generative adversarial nets," in *Advances in Neural Information Processing Systems*, pp. 2172–2180, 2016.

[12] J. Donahue, P. Krähenbühl, and T. Darrell, "Adversarial feature learning," *CoRR*, vol. abs/1605.09782, 2016.

[13] R. Winter and D.-A. Clevert, "Ive-gan: Invariant encoding generative adversarial networks," *arXiv preprint arXiv:1711.08646*, 2017.

[14] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of Machine Learning Research*, vol. 9, no. Nov, pp. 2579–2605, 2008.

[15] D. Ulyanov, A. Vedaldi, and V. Lempitsky, "Adversarial generator-encoder networks," *arXiv preprint arXiv:1704.02304*, 2017.

[16] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, *et al.*, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[17] Y. Huang and L. Mucke, "Alzheimer mechanisms and therapeutic strategies," *Cell*,

vol. 148, no. 6, pp. 1204–1222, 2012.
[18] A. Ivakhnenko and V. Lapa, *Cybernetic Predicting Devices*. Jprs report, CCM Information Corporation, 1973.
[19] K. Fukushima, "Neural network model for a mechanism of pattern recognition unaffected by shift in position- neocognitron," *Electron. & Commun. Japan*, vol. 62, no. 10, pp. 11–18, 1979.
[20] J. J. DiCarlo, D. Zoccolan, and N. C. Rust, "How does the brain solve visual object recognition?," *Neuron*, vol. 73, no. 3, pp. 415–434, 2012.
[21] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," tech. rep., California Univ San Diego La Jolla Inst for Cognitive Science, 1985.
[22] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, pp. 2672–2680, 2014.
[23] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3431–3440, 2015.
[24] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International Conference on Machine Learning*, pp. 448–456, 2015.
[25] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. ICML*, vol. 30, 2013.
[26] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.
[27] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.
[28] J. Christopher M.Bishop, M.Jordan, *Pattern Recognition and Machine Learning*. Springer, 2006.
[29] K. Janocha and W. M. Czarnecki, "On loss functions for deep neural networks in classification," *arXiv preprint arXiv:1702.05659*, 2017.
[30] J. A. Suykens and J. Vandewalle, "Least squares support vector machine classifiers," *Neural processing letters*, vol. 9, no. 3, pp. 293–300, 1999.
[31] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.
[32] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, "Theano: A cpu and gpu math compiler in python," in *Proc. 9th Python in Science Conf*, pp. 1–7, 2010.
[33] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, pp. 675–678, ACM, 2014.
[34] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *International conference on machine learning*, pp. 1139–1147, 2013.
[35] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.
[36] J. Perla, "Notes on adagrad,"
[37] M. D. Zeiler, "Adadelta: an adaptive learning rate method," *arXiv preprint*

arXiv:1212.5701, 2012.
[38] G. Hinton, N. Srivastava, and K. Swersky, "Rmsprop: Divide the gradient by a running average of its recent magnitude," *Neural networks for machine learning, Coursera lecture 6e*, 2012.
[39] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.
[40] J. F. Nash *et al.*, "Equilibrium points in n-person games," *Proceedings of the national academy of sciences*, vol. 36, no. 1, pp. 48–49, 1950.
[41] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved techniques for training gans," in *Advances in Neural Information Processing Systems*, pp. 2234–2242, 2016.
[42] J. Wu, C. Zhang, T. Xue, B. Freeman, and J. Tenenbaum, "Learning a probabilistic latent space of object shapes via 3d generative-adversarial modeling," in *Advances in Neural Information Processing Systems*, pp. 82–90, 2016.
[43] C. Vondrick, H. Pirsiavash, and A. Torralba, "Generating videos with scene dynamics," in *Advances In Neural Information Processing Systems*, pp. 613–621, 2016.
[44] J. Bao, D. Chen, F. Wen, H. Li, and G. Hua, "CVAE-GAN: fine-grained image generation through asymmetric training," *CoRR*, vol. abs/1703.10155, 2017.
[45] A. Kumar, P. Sattigeri, and P. T. Fletcher, "Improved semi-supervised learning with gans using manifold invariances," *CoRR*, vol. abs/1705.08850, 2017.
[46] L. Metz, B. Poole, D. Pfau, and J. Sohl-Dickstein, "Unrolled generative adversarial networks," *CoRR*, vol. abs/1611.02163, 2016.
[47] M. Mirza and S. Osindero, "Conditional generative adversarial nets," *arXiv preprint arXiv:1411.1784*, 2014.
[48] A. Odena, C. Olah, and J. Shlens, "Conditional image synthesis with auxiliary classifier gans," *arXiv preprint arXiv:1610.09585*, 2016.
[49] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, "Image-to-image translation with conditional adversarial networks," *arXiv preprint arXiv:1611.07004*, 2016.
[50] M. Arjovsky and L. Bottou, "Towards principled methods for training generative adversarial networks," *arXiv preprint arXiv:1701.04862*, 2017.
[51] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. Courville, "Improved training of wasserstein gans," *arXiv preprint arXiv:1704.00028*, 2017.
[52] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein gan," *arXiv preprint arXiv:1701.07875*, 2017.
[53] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *arXiv preprint arXiv:1312.6114*, 2013.
[54] A. B. L. Larsen, S. K. Sønderby, H. Larochelle, and O. Winther, "Autoencoding beyond pixels using a learned similarity metric," *arXiv preprint arXiv:1512.09300*, 2015.
[55] H. Zhang, T. Xu, H. Li, S. Zhang, X. Huang, X. Wang, and D. Metaxas, "Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks," *arXiv preprint arXiv:1612.03242*, 2016.
[56] M. Noroozi and P. Favaro, "Unsupervised learning of visual representations by solving jigsaw puzzles," in *European Conference on Computer Vision*, pp. 69–84, Springer, 2016.
[57] X. Wang and A. Gupta, "Unsupervised learning of visual representations using videos," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 2794–2802, 2015.
[58] C. Doersch, A. Gupta, and A. A. Efros, "Unsupervised visual representation learning by context prediction," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1422–1430, 2015.

[59] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired image-to-image translation using cycle-consistent adversarial networks," *arXiv preprint arXiv:1703.10593*, 2017.

[60] T. Kim, M. Cha, H. Kim, J. Lee, and J. Kim, "Learning to discover cross-domain relations with generative adversarial networks," *arXiv preprint arXiv:1703.05192*, 2017.

[61] M. Fornasier and H. Rauhut, "Compressive sensing," in *Handbook of mathematical methods in imaging*, pp. 187–228, Springer, 2011.

[62] S. Ji, Y. Xue, and L. Carin, "Bayesian compressive sensing," *IEEE Transactions on Signal Processing*, vol. 56, no. 6, pp. 2346–2356, 2008.

[63] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning," in *NIPS workshop on deep learning and unsupervised feature learning*, vol. 2011, p. 5, 2011.

[64] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," 2009.

A

Derivation for Equation (4.5)

Suppose p is the density of a normal random variable with mean μ_1 and variance ${\sigma_1}^2$, and q is the density of a normal random variable with mean μ_2 and variance ${\sigma_2}^2$. Then KL divergence from p to q is:

$$\int \left[\log(p(x)) - log(q(x))\right] p(x)dx \tag{A.1}$$

$$= \int \left[-\frac{1}{2}\log(2\pi) - \log(\sigma_1) - \frac{1}{2}\left(\frac{x-\mu_1}{\sigma_1}\right)^2 + \frac{1}{2}\log(2\pi) + \log(\sigma_2) + \frac{1}{2}\left(\frac{x-\mu_2}{\sigma_2}\right)^2 \right] \tag{A.2}$$

$$\times \frac{1}{\sqrt{2\pi}\sigma_1} \exp\left[-\frac{1}{2}\left(\frac{x-\mu_1}{\sigma_1}\right)^2 \right] dx \tag{A.3}$$

$$= \int \left\{ \log\left(\frac{\sigma_2}{\sigma_1}\right) + \frac{1}{2}\left[\left(\frac{x-\mu_2}{\sigma_2}\right)^2 - \left(\frac{x-\mu_1}{\sigma_1}\right)^2 \right] \right\} \tag{A.4}$$

$$\times \frac{1}{\sqrt{2\pi}\sigma_1} \exp\left[-\frac{1}{2}\left(\frac{x-\mu_1}{\sigma_1}\right)^2 \right] dx \tag{A.5}$$

$$= E_1 \left\{ \log\left(\frac{\sigma_2}{\sigma_1}\right) + \frac{1}{2}\left[\left(\frac{x-\mu_2}{\sigma_2}\right)^2 - \left(\frac{x-\mu_1}{\sigma_1}\right)^2 \right] \right\} \tag{A.6}$$

$$= \log\left(\frac{\sigma_2}{\sigma_1}\right) + \frac{1}{2\sigma_2^2} E_1\left\{(X-\mu_2)^2\right\} - \frac{1}{2\sigma_1^2} E_1\left\{(X-\mu_1)^2\right\} \tag{A.7}$$

$$= \log\left(\frac{\sigma_2}{\sigma_1}\right) + \frac{1}{2\sigma_2^2} E_1\left\{(X-\mu_2)^2\right\} - \frac{1}{2} \tag{A.8}$$

Note that:

$$(X-\mu_2)^2 = (X-\mu_1+\mu_1-\mu_2)^2 = (X-\mu_1)^2 + 2(X-\mu_1)(\mu_1-\mu_2) + (\mu_1-\mu_2)^2$$

So Equation (A.8) equates to:

$$= \log\left(\frac{\sigma_2}{\sigma_1}\right) + \frac{1}{2\sigma_2^2}\left[E_1\left\{(X-\mu_1)^2\right\} + 2(\mu_1-\mu_2)E_1\left\{X-\mu_1\right\} + (\mu_1-\mu_2)^2\right] - \frac{1}{2} \tag{A.9}$$

$$= \log\left(\frac{\sigma_2}{\sigma_1}\right) + \frac{\sigma_1^2 + (\mu_1-\mu_2)^2}{2\sigma_2^2} - \frac{1}{2} \tag{A.10}$$

If the latent vector has M dimension, and the mean and variance for each dimension is denoted as μ_i and σ_i, and $\mu_2 = 0, \sigma_2 = 1$, then, we can have the KL divergence from $E(x)$ to a unit Gaussian is below:

$$KL(E(x)||N(0,1) = -\frac{M}{2} + \frac{1}{M}\sum_{i=1}^{M}\frac{{\sigma_i}^2 + {\mu_i}^2}{2} - log(\sigma_i) \tag{A.11}$$

where μ_i is the mean of ith dimension on the latent vector, σ_i is the variance of the latent vector.