5-2018

A Continuous Space Generative Model

Erzen Komoni
University of Arkansas, Fayetteville

A Continuous Space Generative Model

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science

by

Erzen B. Komoni
University of Prishtina ”Hasan Prishtina”
Bachelor of Science in Computer Science, 2014

May 2018
University of Arkansas

This thesis is approved for recommendation to the Graduate Council

Michael S. Gashler, Ph.D.
Thesis Director

Xintao Wu, Ph.D.
Committee Member

John Michael Gauch, Ph.D.
Committee Member

Abstract

Generative models are a class of machine learning models capable of producing digital images with plausibly realistic properties. They are useful in such applications as visualizing designs, rendering game scenes, and improving images at higher magnifications. Unfortunately, existing generative models generate only images with a discrete predetermined resolution. This paper presents the Continuous Space Generative Model (CSGM), a novel generative model capable of generating images as a continuous function, rather than as a discrete set of pixel values. Like generative adversarial networks, CSGM trains by alternating between generative and discriminative steps. But unlike generative adversarial networks, CSGM uses only one model for both steps, such that learning can transfer between both operations. Also, the continuous images that CSGM generates may be sampled at arbitrary resolutions, opening the way for new possibilities with generative models. This paper presents results obtained by training on the MNIST dataset of handwritten digits to validate the method, and it elaborates on the potential applications for continuous generative models.

Contents

Chapter 1

Introduction

Many machine learning tasks can be reduced to variations of function estimation. The principal task in function estimation is to learn from a set of sampled observations, and produce a model capable of making novel predictions. The task of function estimation may be divided into two general categories: discrimination and generation. Traditional discriminative models approximate a function that consumes high-dimensional observations (such as digitals images), and produces a low-dimensional descriptive label or classification. That is, if $\boldsymbol{X} = <\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n>$ is a set of high-dimensional feature representations, and $\boldsymbol{Y} = <\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots, \boldsymbol{y}_k>$ is a corresponding set of low-dimensional labels, then a discriminative model would be trained using this data to map from $\boldsymbol{X} \rightarrow \boldsymbol{Y}$. Then, novel feature vectors could be digested by this model to produce meaningful label predictions. By contrast, generative models, which are the focus of this work, operate in the opposite direction. They consume a small description and produce a high-dimensional sample, presumably one that would be considered to belong among the observations that were used to train the model. Generative models have been used for applications such as generating realistic images images [8] or extracting essential features from a dataset [13].

Unfortunately, currently available generative models have several limitations. Many of them are complex and costly or even intractable to train, they tend to be unstable, and they are almost always limited to produce images with a predetermined resolution. In this paper, we present the Continuous Space Generative Model (CSGM), a novel approach for generative modeling that addresses these challenges. CSGM is implemented with a single neural network that trains within a reasonable amount of time using simple stochastic gradient descent. It produces stable results, and it operates in continuous space for both its input and its output values.

The remainder of this paper is layed out as follows: The next section, presents related work in the domain of generative models. Section 3 describes CSGM with detailed pseudocode, algorithm description, and implementation details. Section 4 presents experimental results with the MNIST dataset to validate CSGM. Finally, we discuss new possibilities for ongoing research that is enabled by the introduction of this work.

Chapter 2

Related work

The generative discipline in machine learning has gained much attention from researchers in recent years by showing results in various tasks that were not previously imaginable. These results include the generation of novel yet plausible images that differ from any that the model had been trained on [8][1][10], image-colorization [2], and super-resolution [10]. Deep networks and generative models have been used for planning applications by predicting game objects and state [16], as well as filling missing entries in datasets [5].

To distinguish how generative models represent learned features, we divide them into two main frameworks: implicit and observable. Implicit generative models are a type of neural networks that, within their weights, capture and represent the features learned through observations. They later use those trained weights to generate new samples. The values generated by implicit generative models are not meant for human consumption. They are typically incomprehensible to humans because they are not constrained to conform to any metrics that we would typically understand. Observable generative models extract the feature representation of the data by modeling extrinsic tables which they use later to augment the input for the generation process. These models are considered more transparent. The nature of the data we train on also imposes another way to classify generative models. When the model learns only by processing the entry values without appropriate labeling, this is called unsupervised learning. These models, on the generative domain, have shown extraordinary results by being able to create realistic images, but they lack the ability to be controlled for the desired output class. The other category of learning type is the supervised learning. On this technique we set the target label, and we train the model to fit the output to the given target label. In generative models, however, we see that a combination of these two learning approaches results in more significant output. We provide the model with some relevant information about the category of the output we want and also leave opportunity for the model to create variations of the entries. These models are called conditional generative models.

2.1 Autoencoders

In classic AutoEncoders, a neural network is trained to predict the same features that were given as its input. That is, the network seeks to learn to approximate the identity function $\boldsymbol{X} \to \boldsymbol{X}$. However, the network is constrained to pass the information through a smaller hidden layer (bottleneck). This forces the network to learn to both encode the input in a low-dimensional manner, and then decode it with the other side of the network. This model learns on the first part of the network (the encoder) to select essential features of the data and on the second part (the decoder) to recreate the original image from some intrinsic representation. Autoencoders offer limited use as generative models because they do not necessarily represent their intrinsic encoding with any known probability distribution, causing difficulty for sampling from the internal encoding. Nevertheless, in many cases, Autoencoders have been shown to be very versatile, and are used for compression and as generators for creating new samples. To use these capabilities, for example, we take away the first part of the model and feed new encodings directly into the second part by sampling through some prior probability distribution. The model is then capable of creating new samples from the supplied encoding. With the same part of the model, we are able to extraction the internal representation and the data it processed during the training. This property of the model is used mostly for compression.

Variational AutoEncoders (VAE) [13] are a variation of classic Autoencoders which train to maximize the lower bond of the log-probability of a prior distribution. VAEs have a similar approach to traditional autoencoders, except that here the encoder is regularized in a manner that promotes encodings that are distributed according to a Normal distribution. This process is formally described in the following equation:

$$l_i(\theta,\phi) = -E_{z\ q_\theta(z|x_i)}[log\ p_\phi(x_i|z)] + KL(q_\theta(z|x_i)||p(x))$$

where the first part of the equation fits the training data and the second part, normalizes the data distribution on the latent variables by measuring the KL divergence which calculates the difference between distributions. For creating new images from VAEs, we only use the decoder part of the network by sampling different values from the distribution of the latent variables.

2.2 PixelRNN

Another generative model, called PixelRNN, takes advantage of a recurrent neural networks (RNNs) sequential prediction capabilities to calculate the products of joint conditional probabilities at the pixel level, and PixelCNN (a variation that uses convolutional layers instead of recurrent layers) calculate the joint probability for every pixel as conditioned on all proceeding pixels [17]. This probability product is shown by the following formula:

$$p(\boldsymbol{x}) = \prod_{i=1}^{n^2} p(x_i \mid x_1, \ldots, x_{i-1})$$

The generation flow of PixelRNN/CNN goes from leftmost-top pixel looping through each row. For every pixel $\boldsymbol{x_i}$ of every row in the picture, it calculates the probability $p(x_i \mid x_1, \ldots, x_{i-1})$ conditioned on all the previous pixels of the upper rows and the left pixels on the same row. These models have a multinomial output which computes the pixel as discrete values through a softmax output layer. The pixel value is represented as a probability on the 255 red-green-blue scheme. The probability of the blue color is conditioned to the probability of the green which is also conditioned on the probability of the red color for the same pixel. In the paper by Oord et al. [17], there are presented two new types of layers in the two-dimensional space. These layers extend the LSTM [9] architecture that gives gated outputs on RNNs. The first type of layer presented in that paper is Row LSTM which applies the convolution on each row by computing the value of each pixel from top and left pixels. Diagonal BiLSTM computes the pixel value diagonally from the diagonal-left and left pixel. In deeper architectures, PixelRNNs face the problem of fading gradient. To overcome this problem, they implement residual connections to propagate the error on further layers [17]. Other variants of this architecture were introduced later. Conditional PixelCNN [19] aims to create a more oriented outcome by providing the model with prior information.

2.3 Generative Adversarial nets

A class of generative models that has recently become very popular is Generative Adversarial Networks (GANs). GAN is a framework with two neural network models that implement a mini-max zero-sum game [8] by making the two networks compete against each other. The first network is called the generator (G) which trains to fit the data probability distribution $\boldsymbol{p_x(x)}$ so that it can trick the other network, the discriminator (D), by producing realistic like images. The discriminators role is to detect if the image comes from the real dataset or is being created by the generators. This process is described mathematically by the following equation:

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

Over time, the two networks become better at performing their role. The generator produces a better output which lies closer to the $\boldsymbol{p_x(x)}$ distribution, and the discriminator improves in distinguishing if the input comes from the real dataset or the generator. The training is considered to have converged when the global optimum is reached. That occurs when the discriminator is no longer able to distinguish the input produced by the generator. When these probability distributions become indistinguishable by the discriminator, $\boldsymbol{p_G = p}_{data}$, it predicts 50% of being from one input or another $\boldsymbol{D_G(x)} = \frac{1}{2}$ [8]. Even though this architecture produces astonishing output, in the original architecture it has been observed that it is hard to keep the model stable during training and that the results lack in properly representing the global structure. The introduction of GANs by Goodfellow et al. [8] in 2014 triggered a large amount of research in the field. Since then, many variations and training improvements of this architecture have been introduced [1][18][3][10]. Other approaches have taken the GAN architecture and implemented it in sequential generation models. PixelGAN Autoencoders [14] take a similar approach based on distribution priors.

2.4 Conditional Generative Models

Besides being slow to train, generative neural networks also have the problem of not allowing users to specify what they should generate while still allowing it some degree of freedom in generating quality results. To achieve this balance, the model can be provided with input that specifies a label or conditional value that directs the nature of the generated output. In the work of Zhang et al. [21], they showed the capabilities of generative models by only serving the model with text description about the properties of the entities they wanted to be generated, and the model output created images that were clearly relevant to the specifed labels.

2.5 Dimensionality reduction

In high resolution images, the total number of pixels is typically in the order of many millions. This can cause significant computational cost for processing or otherwise operating with such images. A conventional approach to speeding up the computation is to implement some dimensionality reductions on the input data and be able to adequately represent the essential features after the reduction. The goal of dimensionality reduction is to be able to represent the same features with less amount of data. This will result in faster computation. The work of Gashler et al. [6] shows that TNLDR technique is able to successfully reduce the dimensions from observed pictures. From the intrinsic representation of the observed data, the model is able to recreate the state of a dynamic system that was observed. The aforementioned data dimensionality reduction principle is also used in this paper to capture essential features of data which feeds into the model to condition the output.

Chapter 3

Continuous Space Generative Model

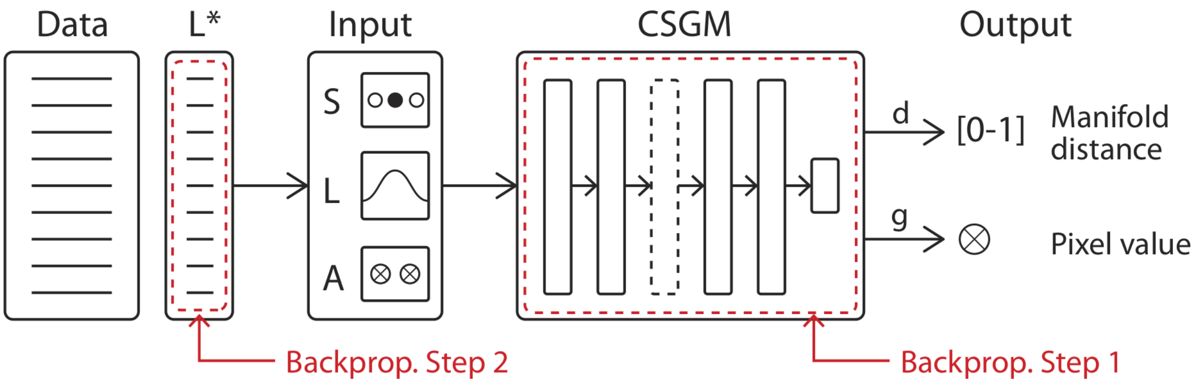

Figure 3.1: CSGM Architecture

This section describes the Continuous Space Generative Model (CSGM). CSGM is implemented with a single neural network model. It operates in a space of both continuous inputs and continuous output.

The observation process of CSGM follows the micro-perception principle, which can be described as using a single pixel camera. It is trained by randomly visiting points on the image. Pixel values are considered to be a function of that pixel's position, so the pixel coordinates are treated as input features, and the pixel values are used as labels for training. Because pixels are modeled as a continuous function, the pixel values at any coordinates, including sub-pixel coordinates, may be estimated after training by simply feeding those coordinates through the model. Previous work from Gashler et al. [6] demonstrated that this principle could yield reasonable results for modeling dynamical systems. This showed not only that useful results can be obtained by exploring the environment one point at a time, but also that we can intrinsically capture essential features in this manner, and recreate anticipated observations.

This paper shows that micro-perception can be adopted in the generative domain to generating images in the form of a continuous function mapping from coordinates to pixel values. As can be observed in equation 3.1, models that represent images as continuous functions have the advantage of faster learning because they only process a fraction of the data at a time. The pixel-wise continuity also sets no theoretical limit to the resolution of the generated images, making this a potential research topic for utilizing resources with arbitrarily large or varying sizes.

$$<(t_1, x_{\otimes 1}),\ (t_2, x_{\otimes 2}),\ \ldots,\ (t_n, x_{\otimes n})> \tag{3.1}$$

The primary reason for generating pixel-wise is to facilitate a continuous approach to images and other resources that are traditionally processed as arrays or matrices. If an image were to be represented as a vector for all its pixels $\boldsymbol{X} = <\boldsymbol{x_1},\ \boldsymbol{x_2},\ \ldots,\ \boldsymbol{x_n}>$, then in a finite amount of time $\boldsymbol{T} = <\boldsymbol{t_1},\ \boldsymbol{t_2},\ \ldots,\ \boldsymbol{t_n}>$, a continuous function might process pairs $(\boldsymbol{t}, \boldsymbol{x})$ in sequential time order $<(\boldsymbol{t_1},\ \boldsymbol{x_1}), (\boldsymbol{t_2},\ \boldsymbol{x_2})\ \ldots\ (\boldsymbol{t_n},\ \boldsymbol{x_n})>$. CSGM, however, does this in a stochastic manner. We denote the processing sequence of CSGM in equation 3.1, where $\boldsymbol{x}_{\otimes n}$ is the n-th iteration of processing a randomly chosen pixel.

Figure 3.1 shows the architecture of CSGM. The input for CSGM consists of three vectors, concatenated into one input vector, $\boldsymbol{X} = <\boldsymbol{S},\ \boldsymbol{L},\ \boldsymbol{A}_{\otimes}>$. During training, $\boldsymbol{S}$ represents the label assigned to the sample feature vector $\boldsymbol{x}^{(i)}$. After training, $\boldsymbol{S}$ is the label for which an image should be generated. It is analogous to the conditioning term used in CGANs [15] and other generative semi-supervised models [12]. If an unconditional generative model is desired, $\boldsymbol{S}$ may be an empty vector. $\boldsymbol{L}$ is a noise vector that enables CSGM to generate novel output. We draw this noise from a standard Normal distribution $\mathcal{N}(\mathbf{0},\ \mathbf{1})$. $\boldsymbol{A}_{\otimes}$ is an attention vector that specifies the coordinates of the current pixel of interest. If $\boldsymbol{x}^{(i)}$ is a two-dimensional image, then $\boldsymbol{A}_{\otimes}$ would contain two elements to specify the horizontal and vertical position of a pixel in that image. (The symbol $\otimes$

indicates the point where attention is currently focused. During training these points of attention are chosen randomly. When a new image is generated, the point of attention would raster over all possible coordinates in the image to generate each pixel value one at a time). The CSGM model has two outputs. The first, denoted as $\boldsymbol{d}$, is a scalar value that represents the distance to the underlying probability distribution represented by the training data $\boldsymbol{p_x(x)}$. The second, denoted as $\boldsymbol{g}$, is a vector of channel values for the pixel specified by $\boldsymbol{A}_\otimes$. (For a gray-scale image, this would just be a scalar value indicating pixel brightness. For a color image it would typically be a vector specifying red, green, and blue channel values.)

CSGM resembles several aspects of GANs [8], but it varies in the model and objective function. The objective function in CSGM is simply the sum square error (SSE) $\sum_{i=1}^{n}(y_i - f(x_i))^2$ between the predicted output $\hat{\boldsymbol{y}}$ and the target value $\boldsymbol{y}$. In contrast to a standard GAN which has two neural networks competing against each other in a mini-max zero-sum game [8], CSGM has only one model that performs both functions. This enables learning to transfer between the two tasks.

$$\forall_\otimes\, i \in X_n : \nabla g \leftarrow |y_{\otimes i} - \boldsymbol{CSGM}(<S_n, L, A_{\otimes i}>)|^2 \tag{3.2}$$

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta\ \nabla g \tag{3.3}$$

$$\boldsymbol{L} \leftarrow \boldsymbol{L} + \eta\ \nabla g\ \boldsymbol{w_0} \tag{3.4}$$

CSGM is trained iteratively with stochastic gradient descent. Each iteration may be described with three steps: **First**, a random pixel is selected in one of the training images. Corresponding values for $\boldsymbol{S}$, $\boldsymbol{L}$, and $\boldsymbol{A}_\otimes$ are fed forward through the model to compute estimated values for $\boldsymbol{d}$ and $\boldsymbol{g}$. Then error is computed for these terms as the squared difference with their expected values, and backpropagation is applied to compute a gradient for the model, as given in Equation 3.2. The expected value for $\boldsymbol{g}$ is the actual pixel value in the image from the training data. Ideally, the expected value for $\boldsymbol{d}$ would indicate how close the image (represented in $\boldsymbol{L}$) is to the manifold of good images sampled by the training data. Unfortunately, this value is not known, so we must use a heuristic for training $\boldsymbol{d}$. Since $\boldsymbol{L}$ is initialized randomly, it is unlikely to fall close to the desired manifold. Therefore, we begin by using an expected value of $\boldsymbol{d} = 0$. After several refining iterations, we assume that $\boldsymbol{L}$ has moved onto the desired manifold, as depicted in Figure 3.2, so we linearly increase the expected value for $\boldsymbol{d}$, such that it arrives at the value $\boldsymbol{d} = 1$. The **second** step updates the weights $\boldsymbol{w}$ of the neural network model, as described in equation 3.3. The **third** step of each training iteration refines the latent encoding of the image $\boldsymbol{L}$. As training progresses, each $\boldsymbol{L}$ begins to represent the properties of the corresponding image, and the neural network begins to converge such that all of the training images fall on the manifold where $\boldsymbol{d} = 1$.

The input $\boldsymbol{X}$ is a combination of 3 parts $\boldsymbol{X} = <\boldsymbol{S}, \boldsymbol{L}, \boldsymbol{A}>$ that feed together as a larger vector into the first layer. The compounding subvectors of $\boldsymbol{X}$ have global and local relevance. The state $\boldsymbol{S}$ and latent code $\boldsymbol{L}$ are set globally for each image, whereas the attention $\boldsymbol{A}$ is relevant for every pixel coordinate $\boldsymbol{x}_{\otimes n}$.

State $\boldsymbol{S}$ represents the label $\boldsymbol{y}$ in one-hot encoding which conditions the model to generate a particular type of output. With this parameterization, the model is more focused and produces the desired entities of a target class as argued by Mirza et al. [15]. Prior latent code $\boldsymbol{L}$ is a representation of the image. The primary task when generating an image is to refine $\boldsymbol{L}$ until it falls on the manifold of images sampled by the training data. $\boldsymbol{L}$ is initialized with noise from a normal distribution $\boldsymbol{p_x(x)}$ which is refined in step 3 as described by equation 3.4. Initially, for every entry in the dataset, there is a corresponding $\boldsymbol{L}$ vector that is initialized as described. Throughout the training, these values are refined. With the refinement of $\boldsymbol{L}$, we are able to untangle the noise values and project them to normal distribution space from which we can sample. Also, in $\boldsymbol{L}$ we are able to implement dimensionality reduction in which we capture some important features for each entry and use those as extrinsic feature representations when we generate images with the model [6] [5]. The last part of the input, attention $\boldsymbol{A}$, holds two-pixel coordinate values which are chosen randomly. This serves as pixel value relevance and determines locality.

Our implementation of CSGM utilizes stochastic gradient descent (SGD) for refining its weights. This contrasts with GANs, which typically must be trained with adaptive gradient methods such as Adam [11] because they tend to be too slow and face vanishing gradient with SGD [20].

Algorithm 1 CSGM Training

```
Input: Entry $\boldsymbol{X_i}$, Latent code $\boldsymbol{L_i}$
Output: Refined weights $\boldsymbol{w}$
$\boldsymbol{Y} \leftarrow label(\boldsymbol{X_i})$
$\boldsymbol{m} \leftarrow 0$
for $n : 0 \rightarrow N$ do
    $\boldsymbol{m} = n/N$
    $<\boldsymbol{p}, \boldsymbol{q}> \leftarrow random\_coordinates(\boldsymbol{X_i})$
    $<\boldsymbol{d}, \boldsymbol{g}> \leftarrow Feedforward(<\boldsymbol{p}, \boldsymbol{q}>, \boldsymbol{L_i}, \boldsymbol{Y})$
    $\{grad_{\boldsymbol{w}}, grad_{\boldsymbol{L}}\} \leftarrow \nabla SSE(<\boldsymbol{m}, \boldsymbol{E}_{i(p,\, q)}>, <\boldsymbol{d}, \boldsymbol{g}>)$
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta\ grad_{\boldsymbol{w}}$
    $\boldsymbol{L_i} \leftarrow \boldsymbol{L_i} + \eta\ grad_{\boldsymbol{L}}(\boldsymbol{w_0})$
end for
return $\boldsymbol{w}$
```

CSGM assumes that every dataset of some structured entries has an underlying probability distribution, also known as the manifold of the data, for which the model creates an intrinsic representation. This probability distribution is distributed over the continuous space represented by all possible values for $\boldsymbol{L}$. Within this space, there are divisible regions in which specific output classes fall. This assumption has been shown to be correct in applications that impute missing entries in datasets. The new entries are projected into the manifold space and, as shown in [5], they are classified correctly when cross-checked. The same understanding is used when we create new samples from CSGM. We take $\boldsymbol{K}$ refined latent code values $\boldsymbol{L}$ drawn from some distribution of noise and pass the mean to the model as an input.

The detailed steps of CSGM are described in pseudocode in Algorithm 1 and Algorithm 2. Algorithm 1 describes training process (**first step**) where the weights $\boldsymbol{w}$ of the neural network are updated alongside with the latent code $\boldsymbol{L}$. CSGM iterates randomly in many epochs through all the entries of the dataset and passes them to the method of Algorithm 1. Here, the we initialize the latent code $\boldsymbol{L}$ and $\boldsymbol{m}$. For an arbitrary large number $\boldsymbol{N}$, we iterate through the entry in a pointwise manner. The attention points then are processes to the model and a respective output is generated $<\boldsymbol{d},\boldsymbol{g}>$. These outputs are targeted to desired values of $\boldsymbol{m}$ and pixel value of the entry. The algorithm then computed the gradient and updates the model and latent code through SGD. A more trained model is the result of this step of CSGM.

Algorithm 2 CSGM Generation

```
Input: Label $\boldsymbol{Y}$
Output: Image $<\boldsymbol{X}>$
for K times do
    $\boldsymbol{L_K} \leftarrow \mathcal{N}(0,\ 1)$
    for $n : 0 \rightarrow N$ do
        $<\boldsymbol{p}, \boldsymbol{q}> \leftarrow random\_coordinates(\boldsymbol{E})$
        $<\boldsymbol{d}, \boldsymbol{g}> \leftarrow Feedforward(<\boldsymbol{p}, \boldsymbol{q}>, \boldsymbol{L_K}, \boldsymbol{Y})$
        $grad_{\boldsymbol{L}} \leftarrow \nabla SSE(<\mathbf{1}, \boldsymbol{g}>, <\boldsymbol{d}, \boldsymbol{g}>)$
        $\boldsymbol{L_K} \leftarrow \boldsymbol{L_K} + \eta\ grad_{\boldsymbol{L}}(\boldsymbol{w_0})$
    end for
end for
$\boldsymbol{L}_{avg} \leftarrow$ average of all $\boldsymbol{L_K}$
for every output coordinate $<\boldsymbol{p}, \boldsymbol{q}>$ do
    $<x_i> \leftarrow Feedforward(<\boldsymbol{p}, \boldsymbol{q}>, \boldsymbol{L}_{avg}, \boldsymbol{Y})$
end for
return $<x_1, x_2, \ldots, x_n>$
```

Algorithm 2 shows how a trained model can be used to generate novel images that are not found in the training set, but still lie on the manifold sampled by the training set. It follows a similar approach, but there are a few key differences. The latent image representations are refined as during training, but the weights of the model are held constant during image generation. Also, because an image is desired that falls exactly on the manifold, the expected value for $\boldsymbol{d}$ is set to a constant value of 1 throughout the process of image generation. Instead of visiting pixel coordinates in random order, it also makes more sense to systematically raster over all of the pixel coordinates in the image. In order to promote stability, we take $\boldsymbol{K}$ samples of latent code and in an arbitrary large number of iterations, and we loop through the model by not providing the entry values. Instead, on this **third step** of CSGM, we only provide the label value $\boldsymbol{Y}$ to the model and make the model refine only the latent code $\boldsymbol{L_K}$. By doing this, we force the extraction of specific features to the latent code from how the model represents different categories of output (here represented by $\boldsymbol{Y}$). By practice we have learned that taking the mean of all these $\boldsymbol{L_K}$ latent codes we get the best results. We then feed the refined $\boldsymbol{L}_{avg}$ alongside with the label $\boldsymbol{Y}$ to produce the ultimate output of the model, the novel images.

CSGM captures the data manifold both intrinsically and extrinsically. The training of the weights creates an intrinsic representation of the underlying probability distribution of the data $\boldsymbol{p_x(x)}$ whereas the updated external values $\boldsymbol{L}$ combine to the model through the input as external feature extracted in step 2.

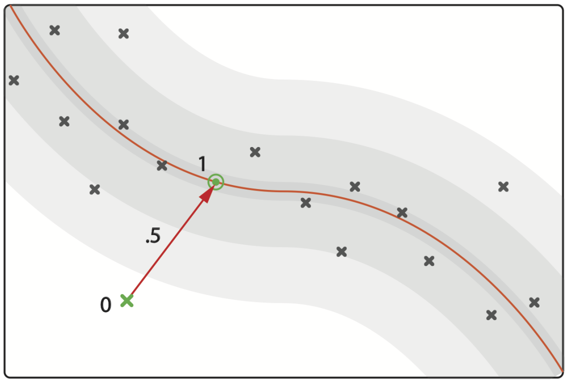

Figure 3.2: CSGM data distribution learning

Chapter 4

Experimental results

In this section, we present a few images generated by CSGM after training on the MNIST dataset of handwritten digits.

The topology of the deep neural network used in this experiment consisted of 10 total layers, alternating between fully connected linear layers of weights followed by leaky rectifier nonlinearities [7]. The input layer, which takes the three sub-vectors combined (as described in Section 3), had 32 input nodes. For these 32 input nodes, the first 10 elements were filled with the value of the subvector $\boldsymbol{S}$. The next 20 values, were taken from the subvector $\boldsymbol{L}$, and the last two values were filled with the attention $\boldsymbol{A}_{\otimes}$ coordinate values. After the input, the first hidden layer had 300 nodes with a non-linear activation function. The second pair of layers with linear and LeakyReLU mapped from 300 to 150 nodes. After that, the layers scaled down from 150 to 100, then to 50 and from 50 to the output layer of 2 output nodes.

Figure 4.1 shows 100 generated images after training on the MNIST dataset. 10 rows of generated digits are shown. Each row contains 10 generated images. The digits in each row were constrained (with the values in $\boldsymbol{S}$) to generate a corresponding digit. These images are not drawn from the training set, but were generated using CSGM after starting with a randomly initialized $\boldsymbol{L}$ vectors. From these results, it can be observed that CSGM was able to capture the data manifold and generate new samples the model has never seen before.

The input for the model was composed of 3 subvectors. The first part $\boldsymbol{S}$ was a subvector of size 10 in which the label from the MNIST data was represented with a one-hot encoding. The second part of the input, the latent values, was a vector of size 20, initially drawn from a Normal distribution $\boldsymbol{L} \sim \mathcal{N}(\mathbf{0},\ \mathbf{1})$ and refined as described in the previous section. The last part of the input

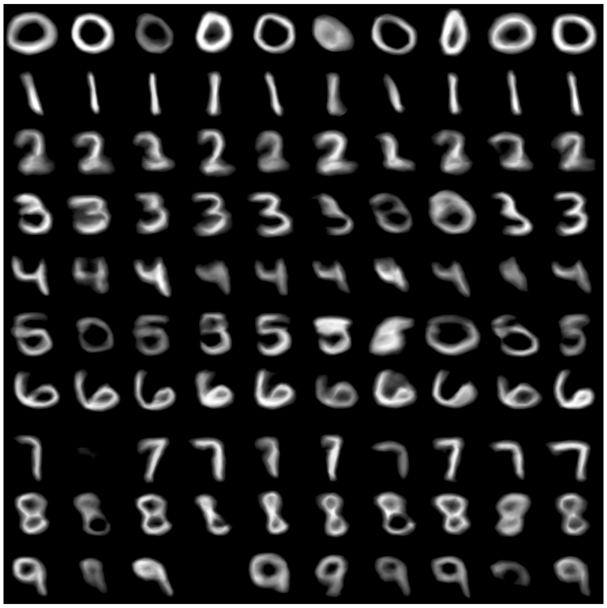

Figure 4.1: MNIST digits generated by CSGM

vector consisted of attention vectors $\boldsymbol{A} = <\boldsymbol{x}, \boldsymbol{y}>$, which were the pairs of coordinates randomly observed from the data. The input was served with the digit label (number) as the condition and the appropriate latent values $\boldsymbol{L}_{i\otimes}$.

For the training process, we set the learning rate to be $\boldsymbol{\eta = 0.01}$, and no momentum was used. We used $\boldsymbol{K = 2}$ for the number of latent codes in step 3 of training (line 13 of the pseudocode in Algorithm 1). All hidden layers were fully connected. The running time in this experiment was 12 hours, and it was executed on a Linux Operating system with 64-bit architecture. The machine processor was Intel i7-4790, 3.60GHz with 8 cores. Total memory of the machine was 32 GB. The code was written in C++ and was built using the Waffles Machine Learning toolkit [4].

It can be observed in Figure 4.1 that there is significant diversity in the shape of the digits, notably different representation of the same digits. This shows that the model captured the underlying manifold of the dataset without simply memorizing individual training patterns. Thus, we can generate new samples from the distribution represented in the training dataset. In Figure 4.2,

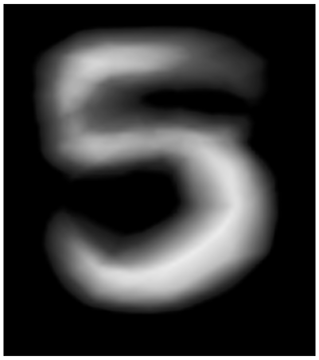

Figure 4.2: Number 2 generated in CSGM scaled programatically 10 times (original MNIST digit 28x28px, generated 300x300px)

we present a single digit taken from the trained model and programmatically scaled it 10 times to its original size. This larger image demonstrates that the generated images are actually continuous, not pixel-based, and can therefore be sampled at arbitrary resolutions. By seeing this example, we can observe that the model conceptualized the representation in real continuous space, and we only need to increase the space size to potentially get arbitrary large scaled examples.

As an analogy between traditional generative models and CSGM, traditional generative models produce a discrete set of pixels like those in a bitmap graphic. By contrast, CSGM generates continuous images, like those illustrated with vector graphics. In the presented images, we can see that the majority of the pictures are easily identifiable and represent a digit. In some examples, however, we see a rather empty space or some representation that looks like two merged numbers (as in the cases of 3 and 8). Some parts of the digits tend to be blurry and fade away continually. We believe this comes from the continuous nature of representation. The randomness of the initial latent code $\boldsymbol{L}$, can cause the fading of gradient if they fall far from the optimal manifold, and they diverge during training of step 2.

Chapter 5

Conclusion and Future work

In this paper, we introduced the Continuous Space Generative Model (CSGM), a new neural network architecture in the generative domain that operates entirely in continuous space. CSGM takes advantage of traditional optimization techniques like stochastic gradient descent to refine the weights. In CSGM we also demonstrated the use of micro-perception principle and pointwise generation as an approach to treat input and output as continuous functions. This work showed that a single model architecture of a nonlinear multilayer perceptron could be trained through micro-perception to capture the underlying data manifold through continuously scanning individual points. The generated images presented in the previous section are images created by sampling the intrinsic manifold representation from the weights $\boldsymbol{w}$ of the model in conjunction with the extrinsic feature dimensionality reduction represented in $\boldsymbol{L}$.

The results presented in Experiments section are initial and serve as a proof of concept. Certainly, more datasets should be tested in future work to fully evaluate the extent of the generative capabilities of CSGM. However, the results given with the MNIST dataset are sufficient to demonstrate continuous generative models are possible. From the generated images, we see that CSGM successfully captured the global structure of the image but lacked on the sharpness and the details of the images. Therefore refinements to this general algorithm are probably needed before it would have significant utility in real-world applications. However, this work showed that it is possible to achieve satisfactory results without having to resort to a computationally expensive generative model that produces all of the pixels in an image at once. This is especially useful when the dataset consists of very large images.

As was stated previously in this paper, CSGM resembles several aspects of GANs [8]. Further research could be conducted in the direction of contrasting CSGM with GANs. For example, the current architecture could potentially be adjusted to utilize two separate neural networks that play the mini-max zero-sum game between them while following the micro-perception principle and working in continuous space.

References

[1] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein generative adversarial networks. In *International Conference on Machine Learning*, pages 214–223, 2017.

[2] Yun Cao, Zhiming Zhou, Weinan Zhang, and Yong Yu. Unsupervised diverse colorization via generative adversarial networks. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 151–166. Springer, 2017.

[3] Antonia Creswell, Tom White, Vincent Dumoulin, Kai Arulkumaran, Biswa Sengupta, and Anil A Bharath. Generative adversarial networks: An overview. *IEEE Signal Processing Magazine*, 35(1):53–65, 2018.

[4] Michael Gashler. Waffles: A machine learning toolkit. *Journal of Machine Learning Research*, 12(Jul):2383–2387, 2011.

[5] Michael S Gashler, Michael R Smith, Richard Morris, and Tony Martinez. Missing value imputation with unsupervised backpropagation. *Computational Intelligence*, 32(2):196–215, 2016.

[6] Mike Gashler and Tony Martinez. Temporal nonlinear dimensionality reduction. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*, pages 1959–1966. IEEE, 2011.

[7] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 315–323, 2011.

[8] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[9] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[10] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of gans for improved quality, stability, and variation. *arXiv preprint arXiv:1710.10196*, 2017.

[11] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[12] Diederik P Kingma, Shakir Mohamed, Danilo Jimenez Rezende, and Max Welling. Semi-supervised learning with deep generative models. In *Advances in Neural Information Processing Systems*, pages 3581–3589, 2014.

[13] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[14] Alireza Makhzani and Brendan J Frey. Pixelgan autoencoders. In *Advances in Neural Information Processing Systems*, pages 1972–1982, 2017.

[15] Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784*, 2014.

[16] Junhyuk Oh, Xiaoxiao Guo, Honglak Lee, Richard L Lewis, and Satinder Singh. Action-conditional video prediction using deep networks in atari games. In *Advances in Neural Information Processing Systems*, pages 2863–2871, 2015.

[17] Aaron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks. *arXiv preprint arXiv:1601.06759*, 2016.

[18] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. In *Advances in Neural Information Processing Systems*, pages 2234–2242, 2016.

[19] Aaron van den Oord, Nal Kalchbrenner, Lasse Espeholt, Oriol Vinyals, Alex Graves, et al. Conditional image generation with pixelcnn decoders. In *Advances in Neural Information Processing Systems*, pages 4790–4798, 2016.

[20] Ashia C Wilson, Rebecca Roelofs, Mitchell Stern, Nati Srebro, and Benjamin Recht. The marginal value of adaptive gradient methods in machine learning. In *Advances in Neural Information Processing Systems*, pages 4151–4161, 2017.

[21] Han Zhang, Tao Xu, Hongsheng Li, Shaoting Zhang, Xiaolei Huang, Xiaogang Wang, and Dimitris Metaxas. Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks. In *IEEE Int. Conf. Comput. Vision (ICCV)*, pages 5907–5915, 2017.