Generative Mixture of Networks

Ershad Banijamali[1] Ali Ghodsi[2] Pascal Popuart[1]
[1]School of Computer Science, University of Waterloo
[2]Department of Statistics and Actuarial Science, University of Waterloo
{sbanijam , aghodsib , ppoupart}@uwaterloo.ca

***Abstract*—A generative model based on training deep architectures is proposed. The model consists of K networks that are trained together to learn the underlying distribution of a given data set. The process starts with dividing the input data into K clusters and feeding each of them into a separate network. After few iterations of training networks separately, we use an EM-like algorithm to train the networks together and update the clusters of the data. We call this model Mixture of Networks. The provided model is a platform that can be used for any deep structure and be trained by any conventional objective function for distribution modeling. As the components of the model are neural networks, it has high capability in characterizing complicated data distributions as well as clustering data. We apply the algorithm on MNIST hand-written digits and Yale face datasets. We also demonstrate the clustering ability of the model using some real-world and toy examples.**

INTRODUCTION

Deep architectures have shown excellent performance in various tasks of learning including, but not limited to, classification and regression, dimension reduction, object detection, and voice recognition. In this work, we focus on another task, which is building a generative model. Generative models are used to characterize the underlying distribution of the data and then randomly generate samples according to their estimation of the distribution. Recently, use of deep architectures in the area of generative models is very popular among researchers.

Related Works

A fundamental work on deep generative models has been done by Hinton et al. [14], where they introduced a fast algorithm for unsupervised training of deep belief networks (DBNs), which are deep graphical models. In a recent work by Salakhutdinov [23], a comprehensive review over this model is presented. Built upon this model, Lee et al. [17] presented a similar network with convolutional layers. They introduced probabilistic max-pooling technique and constructed a translation-invariant model. In [22], another generative model based on DBNs was presented, which was used for image feature extraction. Unsupervised deep representation learning techniques have been used in [2] to build a generative model that can exploit high-level features to generate high-quality samples.

Two important and recent classes of deep generative models are generative adversarial networks (GANs) [10] and variational auto-encoders (VAE) [16]. GANs are trained based on solving a minimax problem to generate samples that are not distinguishable from the samples in the training sets. Based on the variational inference concept, VAEs are designed for fast training and having explicit expression for posterior probability of the latent variable. Many recent advancements in the area of deep generative models are based on these two models [20, 3, 5, 25].

Different types of neural networks have been used to work as a generative model for different applications. In [11], inspired by a human vision system, Recurrent Neural Networks (RNNs) are trained for generating images. [4] proposed a method for training Convolutional Neural Networks (CNNs) for this purpose. In [6], authors trained a deep neural network in a supervised way to be able to generate images of different objects given their names, locations, and angles of view.

In almost all of these works, the probability distribution of the output of the networks do not have a well-structured form. So, we have limited ability to extend these models and build a mixture model based on them. In mixture of expert models [30, 27], we have networks as the component of a mixture of *discreminative* models, where we assume some specific probability distribution on the output of the networks (i.e. Gaussian). However, such assumption for *generative* models, where the output of the network is very high-dimensional, is not practical.

Contribution

In this work, we introduce an algorithm for training mixture of generative models, which consists of deep networks as its components. Instead of using the whole training dataset to train one single network, our model is based on training multiple smaller networks by clusters of data. By smaller networks, we mean networks with fewer number of nodes in hidden layers, compared to a network trained by the whole dataset. All the above-mentioned models can be components of this work to build a generative mixture model. The proposed method works under the assumption that components of a mixture model do not provide a closed form expression for the probability distribution of their output. We provide an algorithm which is inspired by expectation maximization (EM) to overcome this challenge.

There are multiple advantages for the proposed algorithm compared to its predecessors, including:

- The accuracy of the output samples is higher, as each network is trained only with similar data points.
- After training the model, users can decide the *category* of data they want to generate, instead of randomly generating samples.

- The model, like other mixture models, can be used as a clustering method.

In the next section, the general idea of mixture models is shortly overviewed. Then, we describe the steps of the algorithm in detail. At last, the performance of the proposed algorithm is evaluated, both as a clustering and a generative model.

Background: Mixture Models and EM

Mixture models are used to estimate the probability distribution of a given sample set where the overall distribution consists of several components. For the case of parametric mixture model, distribution of components are presumed to have some parametric form. Let θ denote the parameter set of a mixture model that has K components, i.e. $\theta = \{\theta_1, \theta_2, ..., \theta_K\}$, where θ_j represents the parameters of jth component. For example, if the components are assumed to be Gaussian, then $\theta_j = \{\mu_j, \Sigma_j\}$. A popular way to estimate θ_j's is using the Expectation Maximization (EM) algorithm. In the expectation (E) step of the EM algorithm, the membership probability of each data point is calculated for all components. In fact, it is the posterior probability over the mixture components. Let m_{ij} be the probability of being a member of the jth component given the ith data point, $\mathbf{x}_i$, i.e.:

$$m_{ij} = P(j|\mathbf{x}_i, \theta_j) \tag{1}$$

In the Maximization (M) step of the EM, the parameters of each component are updated using the membership probabilities. Each point contributes to updating the component parameters based on its component membership probability. To optimize the parameters, the E-step and M-step are consecutively taken until the algorithm converges to a local optimum or the maximum number of iterations is reached.

Mixture of Networks

Inspired by the mixture models, we propose a combined generative and clustering algorithm. The learning process is completely unsupervised. Components of our model are constructed by networks. Neural networks have shown strong capabilities in estimating the distribution of complicated datasets. In fact, there is no constraint on the form of the distribution for components, e.g. Gaussian, Poisson, etc. Therefore, there is no parameter θ_j for the components to describe the distribution explicitly. Instead, the parameters of our model are the weights in the networks that introduce an implicit probability distribution at the output of the networks. We denote the set of weights of the jth network by $\mathbf{w}_j$.

We train multiple networks using the training data set. Similar to the EM algorithm, updating the network weights is based on membership probabilities. This means that we would like the points with high membership probability to an specific network to play significant role in further training that network. Therefore, the effect of different data points for updating the parameters of each network will be different. Each network tries to characterize one part of a multi-modal distribution function of the training data. In this section, we describe a mechanism that involves two steps of training and takes this problem into account.

Steps of the training

Step 1: The process starts with clustering the training set into K partitions using a simple and hard-decision clustering algorithm, e.g. k-means. By hard-decision, we mean the algorithm will assign data point $\mathbf{x}_i$ to exactly one cluster with probability one and to the rest of the clusters with probability zero. The knee method [24] can be used to determine the number of clusters. Suppose X is the training set with N data points $X = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\}$ in $\mathbb{R}^D$, and X^j represents the jth cluster in this set, $X = \{X^1; X^2; ...; X^K\}$. The clustering algorithm will divide the distribution of the training data into K parts. Each of these parts contains similar data points and has a smoother behavior compared to the original distribution of the whole data set.

Each cluster of the data is used to train one network, i.e. the jth network is trained by the jth cluster. Therefore, there will be K networks. The structures of the networks, i.e. the number of layers and number of neurons in each layer, are identical. But, as we train networks with different subsets of the training set, parameters of the networks will be different. The ultimate goal for the networks is to minimize their defined cost function by adjusting their parameters. The cost function is a measure of dissimilarity between the training set and generated sample sets of a network. Let us denote the cost function for the jth network by $\mathcal{C}_j$. $\mathbf{w}_j^*$ is the optimum value for the jth network's parameters if and only if:

$$\mathbf{w}_j^* = \arg\min_{\mathbf{w}_j} \mathcal{C}_j(X^j, Y(\mathbf{w}_j)) \tag{2}$$

where $Y(\mathbf{w}_j)$ is the set of samples generated by jth network. Mini-batch stochastic gradient descent (SGD) is used for training to find a local optimum for $\mathbf{w}_j$.

Algorithm 1 Training jth network by hard-decision clusters

```
- Initialize the network parameter $\mathbf{w}_j$ randomly
for $t = 1$ to $T_1$ do
    - Divide $X^j$ randomly into $b$ mini batches of size $\mathcal{B}$.
    for $i = 1$ to $b$ do
        - Choose the $i$th mini batch of $X^j$
        - Generate $\mathcal{B}$ output samples by the $j$th network using random input
        - Update the network parameters
              $\mathbf{w}_j \leftarrow \mathbf{w}_j - \alpha \frac{\partial \mathcal{C}_j}{\partial \mathbf{w}_j}$        (3)
    end for
end for
```

Algorithm 1 describes the steps of the training of jth network using the jth cluster. The input of the network is a p-dimensional vector whose elements are drawn independently from a uniform distribution. The parameters of the networks are initialized randomly. Parameter α in (3) is the learning rate. The training process is done for T_1 epochs of each cluster.

Let $\hat{\mathbf{w}}_j$ denote the parameters of network j after this step of training.

Step 2: After step 1, the output of the networks tends to be similar to their input datasets, which are clusters of the training set. Now, we propose an iterative model that works like the mixture models. It involves a process in which we further train the networks and cluster the training data set together. Clustering in this step is soft-decision, i.e. point i belongs to cluster j with membership probability $m_{ij} \in [0, 1]$. Training the networks is also affected by these probabilities and different points will contribute differently in updating the parameters of the networks. However, instead of making any assumption on the distribution of the model's components, we propose an updating algorithm that is based on the output of trained networks in previous iterations. This means that if a data point is similar to the current outputs of one network, then it will have a high level of contribution in updating the parameters of that network in the next iteration. Note that in step 2 of the algorithm, the whole training set is used to train each network.

To calculate the membership probabilities, we should have the probability distribution function for each component or network. As we did not impose such constraint on our model, we use kernel similarity between the data points and the generated samples of each network. In order to do this measurement, we generate $\mathcal{S}$ samples by each of the networks. $Y^j = Y(\hat{\mathbf{w}}_j) = \{\mathbf{y}_1^j, \mathbf{y}_2^j, ..., \mathbf{y}_{\mathcal{S}}^j\}$ represents the set of samples generated by jth network. Let ℓ_{ij} denote similarity of data point $\mathbf{x}_i$ to the samples in Y^j. Then:

$$\ell_{ij} = p(\mathbf{x}_i | \hat{\mathbf{w}}_j) = \frac{1}{\mathcal{S}} \sum_{r=1}^{\mathcal{S}} k(\mathbf{x}_i, \mathbf{y}_r^j) \tag{4}$$

The kernel that we use here is Gaussian.

The membership probability also needs the prior probability over each component, which is denoted by π_j. The initial value of π_j in step 2 is: $\pi_j = |X^j|/N$. Here, the membership probability is interpreted as the probability that network j has produced data point $\mathbf{x}_i$, and is given by:

$$m_{ij} = \frac{\ell_{ij}\pi_j}{\sum\limits_{k=1}^{K} \ell_{ik}\pi_k} \tag{5}$$

Note that we should have $\sum_{r=1}^{K} m_{ir} = 1$. Value of the prior probabilities after the first iteration in this step is updated by: $\pi_j = (\sum_{i=1}^{N} m_{ij})/N$. Similar to the EM algorithm, we want the effect of point $\mathbf{x}_i$ in updating parameters of network j ($\hat{\mathbf{w}}_j$) to be proportional to m_{ij}. To do so, we multiply the membership probabilities to the learning rate of the SGD algorithm. If a membership is high, then the learning rate will be high and the effect of that point will be high. If the membership probability is low and near zero, the learning rate will be near zero and the algorithm will not update the network parameters based on that point.

Lets call $\boldsymbol{\ell}_i = \{\ell_{i1}, \ell_{i2}, ..., \ell_{iK}\}$ the likelihood vector assigned to the ith data point. Suppose that likelihood of all points for all networks are stored in an $N \times K$ matrix $\mathscr{L}$. Each row of this matrix is corresponding to one point in the training set. Using the above procedure, this matrix is updated iteratively (after using each epoch to update all networks parameters). The initial value of likelihood matrix is obtained by generating $\mathcal{S}$ samples using each of the trained networks in the step 1.

In order to accelerate the learning process, we use mini-batch SGD here, as well. We need to define mini-batch membership. The membership of mini-batch b_i of size $\mathcal{B}$ for the jth network is defined as $m_{b_i,j} = P(j|\hat{\mathbf{w}}_j, \{\mathbf{x}_r \in b_i\})$ and:

$$m_{b_i,j} = \frac{p(\{\mathbf{x}_r \in b_i\}|\hat{\mathbf{w}}_j)\pi_j}{\sum\limits_{k=1}^{K} p(\{\mathbf{x}_r \in b_i\}|\hat{\mathbf{w}}_k)\pi_k} = \frac{\pi_j \prod\limits_{\mathbf{x}_r \in b_i} \ell_{rj}}{\sum\limits_{k=1}^{K} \pi_k \prod\limits_{\mathbf{x}_r \in b_i} \ell_{rk}} \tag{6}$$

For training the network j using b_i, the learning rate is multiplied to $m_{b_i,j}$. According to (6), $m_{b_i,j}$ contains the effect of $\mathcal{B}$ points together. However, these $\mathcal{B}$ points do not necessarily have similar likelihood vectors. So, the multiplication in (6) can mix the effect of important and non-important points for training an specific network. To solve this issue we should somehow put points which are important for training a network together. A systematic solution is to rearrange the rows of the likelihood matrix $\mathscr{L}$ at each iteration of the step 2, such that the first N_1 rows have the maximum likelihood in their first columns, the next N_2 rows have the maximum likelihood in their second columns, and so on. In fact:

$$N_j = |\{\mathbf{x}_i | \ell_{ij} \geq \ell_{ik} \ , \forall k \neq j\}| \tag{7}$$

and obviously $\sum_{k=1}^{K} N_k = N$. The process is similar to the bootstrap sampling. The corresponding data points to the rows of $\mathscr{L}$ are also rearranged in the same way.

Algorithm 2 Training networks using soft-decision clusters

```
- Initialize likelihood matrix $\mathscr{L}$ based on the clusters in Step 1
for $t = 1$ to $T_2$ do
    - Rearrange data set $X$
    - Divide $X$ into $\lfloor \frac{N}{\mathcal{B}} \rfloor$ mini-batches of size $\mathcal{B}$.
    - Compute the mini-batch memberships.
    for $j = 1$ to $K$ do
        Choose $j$th network
        for $i = 1$ to $\lfloor \frac{N}{\mathcal{B}} \rfloor$ do
            - Choose $i$th mini-batch of $X$
            - Generate $\mathcal{B}$ samples by $j$th network
            - Update the network parameters
                $\hat{\mathbf{w}}_j \leftarrow \hat{\mathbf{w}}_j - m_{b_i,j} \times \beta \frac{\partial \mathcal{C}_j}{\partial \hat{\mathbf{w}}_j}$     (8)
        end for
    end for
    - Update the likelihood matrix $\mathscr{L}$
end for
```

Algorithm 2 summarizes the described procedure in the step 2. Rearranging data set X in this algorithm refers to the procedure stated above. Note that dividing the data into mini-batches is not random in the step 2. The whole process of this step is done for T_2 epochs or iterations.

After this step, the training process is finished. Now we have a hyper-network consists of K small networks with similar structures but different parameters. To generate samples randomly using the hyper-network, one of the networks is randomly chosen based on the priors. That is, the jth network is chosen by probability π_j. To generate a sample from a specific cluster, the corresponding network should be picked manually. Then using a random input, the selected network generates the desired sample.

A feature that distinguishes this model from the previous unsupervised generative models is its capability to generate a specific type of sample. For example, if the networks are trained over a set of face images with different expressions, then it can be used to generate a face in a special category (age, expression, illumination, and etc.), e.g. "a laughing old man", instead of generating samples randomly and waiting for our desired output. This can have many applications including automatic visualization of text.

Maximum Mean Discrepancy as the cost function

The proposed structure in this paper can be trained by any conventional objective function at the output (for example the objective in [6]). However, here we use the maximum mean discrepancy (*MMD*), introduced by Gretton et al. [12], because of its simplicity and effectiveness. *MMD* was also used in two recent works [7, 19]. Therefore, the model parameters are learned based on minimizing the distance between the distribution of the samples generated by the network and samples from the training set, using *MMD*.

Suppose $\mathbf{x}$ has distribution p and $\mathbf{y}$ has distribution q. Let $\mathcal{F}$ be a class of functions. The squared population *MMD* is:

$$MMD^2(\mathcal{F},p,q) = \big[\sup_{f\in\mathcal{F}} \big(\mathbf{E}_x[f(\mathbf{x})] - \mathbf{E}_y[f(\mathbf{y})]\big)\big]^2. \tag{9}$$

If $\mathcal{F}$ is a class of functions in the unit ball in a universal Reproducing Kernel Hilbert-Space (RKHS), then *MMD* is zero if and only if $p = q$ [13]. In this case, *MMD* can also be written in the form of a continuous kernel in that RKHS.

However, in our applications, the underlying *pdf* of the sample sets are unknown. Suppose we have two sample sets $X = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_M\}$ and $Y = \{\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_N\}$. The unbiased empirical estimation of the squared *MMD*, according to [13], for these two sets is given as:

$$\begin{aligned} MMD^2(\mathcal{F},X,Y) &= \tfrac{1}{M(M-1)} \sum_{i=1}^{M}\sum_{j\neq i}^{M} k(\mathbf{x}_i,\mathbf{x}_j) \\ &+ \tfrac{1}{N(N-1)} \sum_{i=1}^{N}\sum_{j\neq i}^{N} k(\mathbf{y}_i,\mathbf{y}_j) - \tfrac{2}{MN} \sum_{i=1}^{M}\sum_{j=1}^{N} k(\mathbf{x}_i,\mathbf{y}_j) \end{aligned} \tag{10}$$

We will use Gaussian kernel here too.

Making the Algorithm Faster and More Effective

Our results show that the batch membership can be very small for most of the batches. So, for each step of training of the networks, we only use the batches that have membership probability more than a threshold (in our experiments 0.001) and do not use the rest of the batches that have negligible batch memberships. This way, the training process will be much faster.

In [21], authors have shown that the power of kernel-based methods, such as *MMD*, for two-sample test problem drops polynomially with increasing dimensions. This suggests that a dimension reduction is helpful as a data pre-processing step for high-dimension datasets. Using an autoencoder is a solution here. We train an autoencoder separately using the complete training dataset. The networks in this scenario should be trained using a low-dimensional version of data. At the output of the generative networks, the decoder part of the autoencoder is used to map the data back into the original space. The hard-decision clustering in the first step of our algorithm can be either performed on the original data or its low-dimensional version.

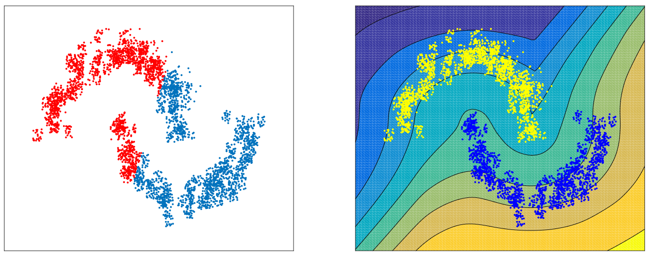
Experiment Results

To highlight the clustering capability of the proposed algorithm, we first apply it on synthetic toy datasets and real-world datasets. Then, we apply the algorithm on two real-world datasets: MNIST hand-written digits dataset and the Yale Face Database. In all of these experiments, the components of the model are fully-connected networks with multiple layers. Input to the networks is a random vector with elements drawn independently from uniform distribution in $[-1, 1]$. The number of layers, number of hidden units in each layer, and the dimension of random input depend on the dataset. The activation function for all hidden layers is ReLU and sigmoid for the output layer. All hyper parameters of the model are set by validation. For each dataset, we keep a portion of data points only for validation. This portion is not used for training. The validation set is also used to prevent overfitting. We continue the training until the average log-likelihood of the validation set is saturated.

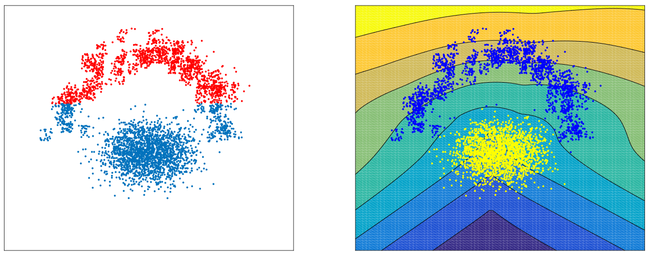
Performance as a clustering model

Toy datasets: In this section, we use three small toy datasets to visualize the clustering performance of the algorithm. We call these datasets two-moon, moon-circle, and two-circle. The first two datasets have 4000 data points and the last one has 4500 data points. All datasets have two dimensions. The datasets are first divided into two parts using k-means and then fed to the model. The model has two networks. We used similar structure for the networks for all datasets. The networks has 3 hidden layers with 32, 128, and 32 hidden units. The input to the network is 2-dimensional. For all of the experiments $T_1 = 30$, $T_2 = 200$, and $\mathcal{B} = 100$.

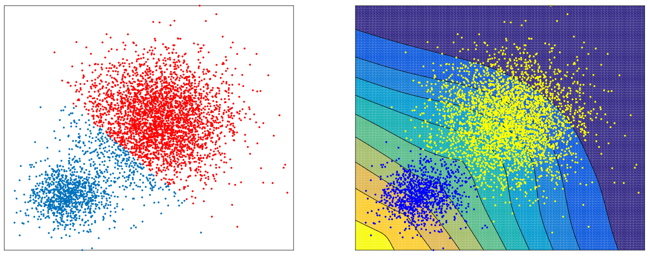
Fig. 1 shows the results of these experiments. As we can see, the algorithm could learn the model parameters to identify the natural clusters. This shows that the algorithm can successfully



(a)



(b)



(c)

Fig. 1: (a), (b), and (c) represent three different datasets. The left figure shows the initial clustering by k-means and the right figure shows the final clusters using the proposed method as well as the contour of membership probability and its isolines.

characterize the distribution of data clusters. Conventional mixture models, such as Gaussian mixture model, obviously fail to identify these clusters for two-moon and moon-circle. Clustering algorithms based on similarity matrices, such as spectral clustering, could have also identified clusters, but they do not possess the generative aspect of our model. Besides, these algorithms usually include an eigen-decomposition step, which is very computationally expensive when it comes to clustering large datasets.

Real-world datasets: Here, we also evaluate the clustering performance of the algorithm for some real-world datasets based on *clustering purity (CP)*. *CP* is defined for a labeled dataset as a measure of matching between classes and clusters. If $\{C^1, C^2, ..., C^L\}$ are L classes of a dataset X of size N, then a clustering algorithm, $\mathcal{A}$, which divides X into K clusters $\{X^1, X^2, ..., X^K\}$ has *CP*$(\mathcal{A}, X)$ as:

$$\mathit{CP}(\mathcal{A}, X) = \frac{1}{N}\sum_{j=1}^{K}\max_i |C^i \cap X^j|. \tag{11}$$

Note that, for our algorithm, we specify the final clusters by assigning each point to the cluster with highest membership probability.

Table I shows the results of clustering for four different datasets. 1) COIL-20: 32×32 images of 20 different objects from different angels. Dataset has 72 images in each class. 2) Reuters-10K: Reuters dataset [9], contains 810000 English news stories in different categories. We followed the same procedure in [29] to obtain 10000 samples from this set in 4 categories. 3) USPS: This dataset contains 16×16 images of hand-written digits. 4) Isolet: This is from UCI repository and contains the spoken alphabet letter from different individuals. Other statistics of the datasets are mentioned in the table. Number of clusters for these experiments are chosen to be equal to number of classes.

We compared the performance of the algorithm with 4 other algorithms. k-means, which is used as the initial clustering for our method as well. The other three algorithms are based on spectral clustering. NCut is the classic spectral clustering, which assigns cluster labels to the data points by running k-means on the eigenvectors of the Laplacian matrix of dataset graph. Local linear approach for data clustering (LLC) [28], assigns cluster labels to each data point based on linear combination of the kernel similarity between that point and its neighbors. Finally, local discriminant models and global integration (LDMGI) [31], which introduces a novel method for the learning Laplacian matrix by employing manifold structure and local discriminant information. LDMGI is designed specially for image clustering and has shown very good performance for clustering. We run experiments 10 times to obtain the results in the table. As we can see, the proposed algorithm achieves the best or near-best results for different datasets. The structure of each individual network is also mentioned in this table.

TABLE I: Comparison of clustering purity (%) for different datasets. The bold numbers show the best results among these algorithms. d = dimensionality of the original space, n = dataset size, p = dimensionality of the low-dimensional dataset using autoencoder, L = # of classes

Dataset	d	n	p	L	k-means	NCut	LLC	LDMGI	Mix. of Nets	Networks Structure
COIL-20	1024	1440	32	20	62.3±3.1	68.4±5.3	67.5±5.1	75.3±4.9	**77.6±3.1**	10-16-256-256-512-32
Reuters-10K	2000	10000	128	4	53.1±2.8	59.3±4.2	57.1±3.9	43.2±3.7	**63.1±4.2**	12-64-256-512-512-128
USPS	256	9298	32	10	64.9±3.6	73.4±6.3	70.1±3.9	**80.5±5.6**	78.3±3.7	10-16-256-256-512-32
Isolet	7797	617	32	26	63.7±2.8	65.7±3.4	69.3±2.7	68.8±3.6	**71.3±3.0**	12-32-256-256-512-32

Performance as a generative model

MNIST dataset: Samples of MNIST set are 28×28 images of hand-written digits. The dataset consists of 60000 training samples and 10000 test samples. We use 5000 samples in the training set for validation and the rest for training the networks.

We first train an autoencoder, which maps the original data to a 32-dimension space. Although, the knee method suggested 12 clusters here, we divided the training set into 10 clusters using k-means to see if we can capture each class by a single network. So, we will have 10 networks to be trained. The networks have 4 hidden layers with 64, 256, 256, and 512 units. Input of the networks is 12-dimensional. Each network is first trained by Algorithm 1 using its corresponding cluster for 30 epochs ($T_1 = 30$). Then using Algorithm 2, the data points membership probabilities and network parameters are updated up to $T_2 = 200$ iterations. Batch size for both steps is 100. We use weight decay as regularization to improve the generalization of the model.

We repeated the whole process of training ten times. Using k-means, the initial value of *CP* on the low dimensional version of the data is 59.2 ± 3.1. After applying our method *CP* goes up to 80.3 ± 4.2, which is close to the state-of-the-art clustering methods on MNIST according to [29].

Fig. 2 shows the samples generated by our model. An evaluation measure that is commonly used for generative models is the average log-likelihood of the test set, also known as Parzen estimation. We generated 10000 samples randomly by the model and fit a Gaussian Parzen Window. We report our model's average log-likelihood of the test set for MNIST as **308** $\pm$ **2.8**. Table II shows a comparison between different methods in terms of average log-likelihood. However, based on [26], this evaluation for generative models can be misleading. In fact, it has been shown that samples generated by a naive methods may achieve higher log-likelihood, even compared to the training data of the generative model.

TABLE II: Average log-likelihood using Parzen window for Different generative models on MNIST dataset DBN: Deep Belief Network, Stacked CAE: Stacked Contractive Auto-Encoder, Deep GSN: Deep Generative Stochastic Network [1], GAN: Generative Adversarial Network, GMMN+AE: Generative Moment Matching Network with Autoencoder, Mixture of Networks: Our model.

MODEL	AVERAGE LOG-LIKELIHOOD
DBN	138 ± 2
STACKED CAE	121 ± 1.6
DEEP GSN	214 ± 1.1
GAN	225 ± 2
GMMN+AE	282 ± 2
MIXTURE OF NETWORKS	**308** $\pm$ **2.8**

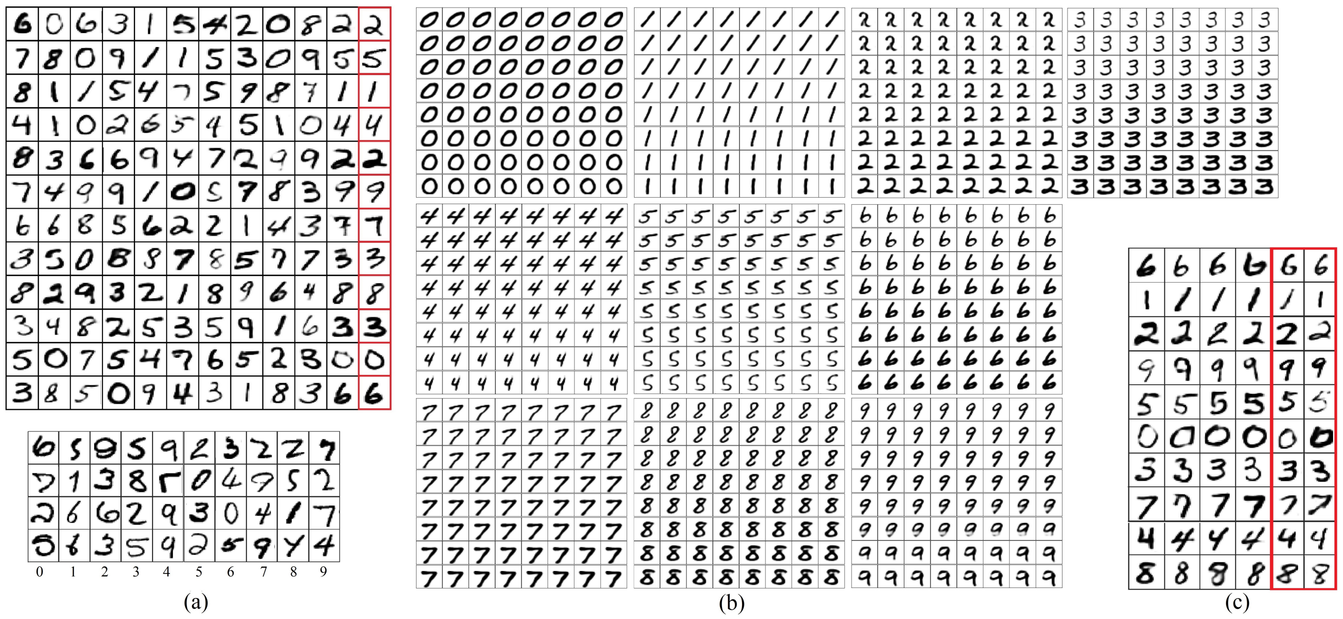


Fig. 2: (a) Top: Samples generated randomly by the hyper-network using the cluster priors. The right most column shows samples from the training set which are nearest-neighbors of their adjacent images in the low-dimension space. This column is added to show that the generated samples are not merely a copy of the training samples. Bottom: Examples of digits of different classes that are mis-clustered by k-means and NCut but mixture of networks clustered them correctly. (b) For each of these 10 sub-images only one networks has been chosen to generate data. For each sub-image we generate two data points with different shapes (top-left corner and bottom-right corner) using two different random inputs. By traversing on an straight line in the latent space we obtain the other data points in the sub-image. As we can see, this shows that the model learns a proper mapping between the latent space and the data space. (c) Digits is each row are generated using one network. Digits in each of the last two columns are generated by giving identical input to different networks.

Face dataset: The other training set we used is the Cropped Extended Yale Face Database B [18, 8]. The dataset contains 2414 near frontal images of 38 individuals under different illuminations. The size of each image is 32×32. We use 214 data points for validation and the rest for training. Using autoencoder dimension is reduced from 1024 to 128. We employ k-means to partition the low-dimension data into four clusters. This number is actually suggested by the knee stability method. Then, Algorithms 1 and 2 are applied to the four networks, consecutively. The networks have 4 hidden layers with 32, 128, 256, and 512 units. For this dataset, the random input is 10-dimensional. $T_1 = 10$ (networks learn the underlying manifold for this data fast) and $T_2 = 100$. The mini batches in both steps of the algorithm contain 120 samples.

Results of the simulations are demonstrated in Fig. 3. Networks produce images in different categories. Categories of data captured by clusters are based on lighting of the images (front lighting, sides lighting, and no lighting). We can also see the smooth changes in the faces when we pick one network and traverse in the latent space. This shows that the networks have learned a proper mapping between the latent space and the real image space.

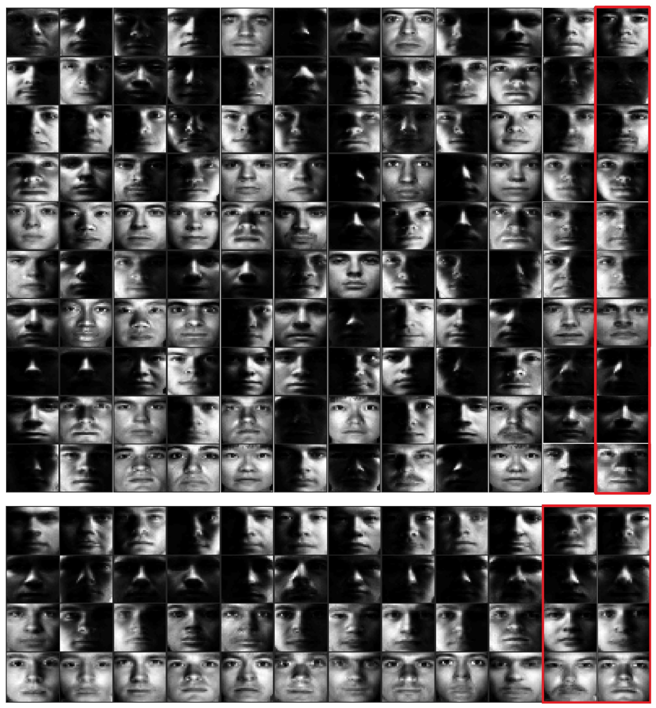


Fig. 3: Top: Samples generated by the hyper-network using cluster priors. The right most column shows samples from the training set which are nearest-neighbors of their adjacent images in the low-dimension space. Bottom: Images generated by networks corresponding to each cluster. Each row is generated by one network. We can see the difference in the generated images which comes from different illuminations. The inputs to the networks for generating each of the last two columns are identical for all networks.

Conclusion and Future work

We proposed an algorithm for developing a generative model using deep architectures. The algorithm has shown advantages compared to the previous generative models, which allows generating and clustering with high accuracy. The efficiency of applying the algorithm on MNIST hand-written digits and the Yale Face Database has been examined, and results support our idea.

It will be specially interesting if a small subset of data is labeled, or when a user has clusters a small portion of data for us and we want to cluster the rest of the data accordingly. In this situation the accuracy of clustering and, consequently, the generative model will increase significantly. One application of this setting is when a hand-written text corpus is given to the model. If we label a small portion of the characters of the corpus and force the algorithm to follow the same rule for clustering the rest of the data, then we can build networks that can mimic handwriting. A related work can be found in [15]. Another direction can be employing the Convolutional Neural Networks, which have shown great performance in vision tasks, instead of fully-connected networks. Then, combining the result by a natural language processing (NLP) model can be interesting. We can convert human language (text or voice) into picture, automatically.

References

[1] Y. Bengio, E. Laufer, G. Alain, and J. Yosinski. Deep generative stochastic networks trainable by backprop. In *Proceedings of The 31st International Conference on Machine Learning*, pages 226–234, 2014.

[2] Y. Bengio, G. Mesnil, Y. Dauphin, and S. Rifai. Better mixing via deep representations. In *Proceedings of The 30th International Conference on Machine Learning*, pages 552–560, 2013.

[3] 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*, pages 2172–2180, 2016.

[4] J. Dai, Y. Lu, and Y.-N. Wu. Generative modeling of convolutional neural networks. *arXiv preprint arXiv:1412.6296*, 2014.

[5] E. L. Denton, S. Chintala, R. Fergus, et al. Deep generative image models using a laplacian pyramid of adversarial networks. In *Advances in neural information processing systems*, pages 1486–1494, 2015.

[6] A. Dosovitskiy, J. Tobias Springenberg, and T. Brox. Learning to generate chairs with convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1538–1546, 2015.

[7] G. K. Dziugaite, D. M. Roy, and Z. Ghahramani. Training generative neural networks via maximum mean discrepancy optimization. *arXiv preprint arXiv:1505.03906*, 2015.

[8] A. S. Georghiades, P. N. Belhumeur, and D. J. Kriegman. From few to many: Illumination cone models for face recognition under variable lighting and pose. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 23(6):643–660, 2001.

[9] Y. Y. R. T. G. GLewis, David D and F. Li. A new benchmark collection for text categorization research. *The Journal of Machine Learning Research*, 2004.

[10] 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*, pages 2672–2680, 2014.

[11] K. Gregor, I. Danihelka, A. Graves, D. Rezende, and D. Wierstra. Draw: A recurrent neural network for image generation. In *Proceedings of The 32nd International Conference on Machine Learning*, pages 1462–1471, 2015.

[12] A. Gretton, K. M. Borgwardt, M. Rasch, B. Schölkopf, and A. J. Smola. A kernel method for the two-sample-problem. In *Advances in neural information processing systems*, pages 513–520, 2006.

[13] A. Gretton, K. M. Borgwardt, M. J. Rasch, B. Schölkopf, and A. Smola. A kernel two-sample test. *The Journal of Machine Learning Research*, 13(1):723–773, 2012.

[14] G. E. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[15] D. P. Kingma, S. Mohamed, D. J. Rezende, and M. Welling. Semi-supervised learning with deep generative models. In *Advances in Neural Information Processing Systems*, pages 3581–3589, 2014.

[16] D. P. Kingma and M. Welling. Auto-encoding variational bayes. In *ICLR*, 2014.

[17] H. Lee, R. Grosse, R. Ranganath, and A. Y. Ng. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 609–616. ACM, 2009.

[18] K.-C. Lee, J. Ho, and D. J. Kriegman. Acquiring linear subspaces for face recognition under variable lighting. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 27(5):684–698, 2005.

[19] Y. Li, K. Swersky, and R. Zemel. Generative moment matching networks. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 1718–1727, 2015.

[20] A. Radford, L. Metz, and S. Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. In *ICLR*, 2016.

[21] A. Ramdas, S. J. Reddi, B. Póczos, A. Singh, and L. Wasserman. On the decreasing power of kernel and distance based nonparametric hypothesis tests in high dimensions. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[22] M. A. Ranzato, J. Susskind, V. Mnih, and G. Hinton. On deep generative models with applications to recognition. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 2857–2864. IEEE, 2011.

[23] R. Salakhutdinov. Learning deep generative models. *Annual Review of Statistics and Its Application*, 2:361–385, 2015.

[24] S. Salvador and P. Chan. Determining the number of clusters/segments in hierarchical clustering/segmentation algorithms. In *Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on*, pages 576–584. IEEE, 2004.

[25] C. K. Sønderby, T. Raiko, L. Maaløe, S. K. Sønderby, and O. Winther. Ladder variational autoencoders. In *Advances In Neural Information Processing Systems*, pages 3738–3746, 2016.

[26] L. Theis, A. v. d. Oord, and M. Bethge. A note on the evaluation of generative models. *arXiv preprint arXiv:1511.01844*, 2015.

[27] S. Waterhouse, D. MacKay, T. Robinson, et al. Bayesian methods for mixtures of experts. In *Advances In Neural Information Processing Systems*, 1996.

[28] M. Wu and B. Schlkopf. A local learning approach for clustering. In *in Proc. NIPS*, pages 1529–1536, 2006.

[29] J. Xie, R. Girshick, and A. Farhadi. Unsupervised deep embedding for clustering analysis. In *Proceedings of the 33rd International Conference on Machine Learning*, 2016.

[30] L. Xu, M. Jordan, and G. Hinton. An alternative model for mxtures of experts. In *Advances In Neural Information Processing Systems*, pages 633–640, 1995.

[31] Y. Yang, D. Xu, F. Nie, S. Yan, and Y. Zhuang. Image clustering using local discriminant models and global integration. *IEEE Transactions on Image Processing*, 19:2761–2773, 2010.