Analyzing 56k font using deep neural network

Ruixiong Shi
Department of Statistical Science
Columbia University
rs3569@columbia.edu

Yuchen Shi
Department of Statistical Science
Columbia University
ys2901@columbia.edu

Abstract—**Generative models play vital roles in machine learning problems. In contrast to discriminative models, generative models can generate synthetic data points by modeling the probability distribution of our data. In our final project, we purposed and implemented a deep neural network in learning the art style from 56 thousand fonts. After learning a 40 dimensional embedding space for different fonts, we visualized the space in both 2D and 3D using T-SNE. Finally, we presented three methodologies in interpolating the embedding space for generating new fonts completely from our network.**
Keywords-component; Deep learning, generative models, generative typography, non-linear dimension reduction, spectrum embedding

I. INTRODUCTION

In past three years, deep learning has been the hot topic and is being applied to different areas, especially in image recognition, natural language processing and natural voice processing. We are interested in a neglected topic but accessible to people's daily life. Everyone has access to a large number of fronts. They are preinstalled in our operating system or provided in some packages online (such as Google Web Fonts Project). Before the digital time and personal computer being widely used everywhere, people write with their own pen and have their own translation. After keyboard and Microsoft word taking care of our entire writing task, we are used to reading Times New Roman or Calibri everyday. Thanks to great Steve Jobs, we can use beautiful fonts for writing but we still miss the personality embedded in our hand-written style. Especially, for Chinese Calligraphy, it is considered as art that reflects Chinese cultural and spirit. Among all kinds of Chinese artwork, Chinese Calligraphy is always considered to have the highest value in auction. Taking Ping An Tie (figure 1) as an example, half of the old imitation (not the original one) was sold for fifty million dollars in 2010.

Figure 1

We think it would be cool if we can design our personal font based on our own hand-written style. However the design usually requires professional typographers who have many years of training and experience. Not to say Chinese has 3000 different commonly used characters. Many people have interest in font appearance and manipulation but don't have the training or time to make the use of professional editing tools. The deep learning curves of font packages act as a barrier. It would great if we can have some application that edit font as easy as those popular image-editing applications.

We are motivated by this and desire to give users without font design background ability to create their own font style, especially to learn the font from great artwork. Very often, we only have less than one hundred words in Chinese Calligraphy artwork. In order to create fonts based on a few hundred words, our algorithm should truly learn the style and structure of chosen artwork and this is when we use neural network. Leon [1] proposed a neural network in 2015 to learn painting style. We want to do similar things for font.

However, after doing some research, we didn't find a Chinese font dataset with desired size (more than 2000 different fonts). We don't want to spend most of our time on collecting dataset in the two-month period, so we choose an English font dataset online and start building our neural network on this dataset. In this paper, we start from introducing some related work on generative topography and we will present the font dataset we used. After that we will explain in detail about the network specification and the techniques we adopt to train the network in detail. Moreover, we will present the software packages and hardware we utilize. Finally, we will conduct some analysis and visualization for the font in a vector space and we will explore methodologies that our network would apply in generating completely new font without human intervention.

II. Related Works

Although there are not many, still some relative works have been done on font learning. Neil in his paper [2] built a generative manifold of standard fonts (figure 2). Every location on that manifold corresponds to a unique and novel typeface. It is obtained by learning a non-linear mapping that intelligently interpolates and extrapolates existing fonts. People can use that manifold to smoothly interpolate and move between existing fonts.

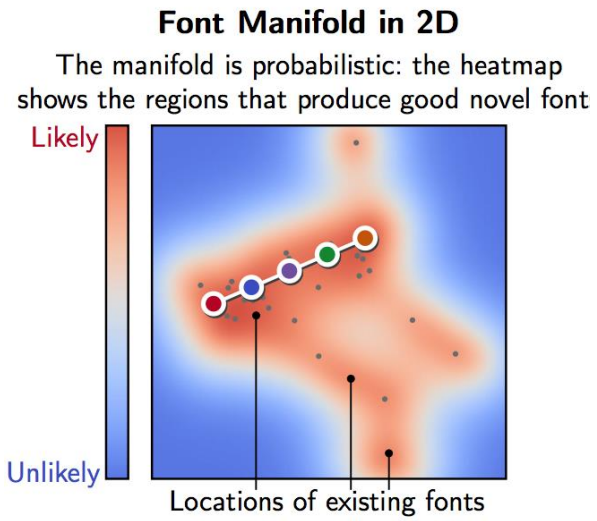


Figure 2

They also provide a standalone Javascript based viewer that allows users to explore both the joint manifold of fonts and manifolds for individual characters. [3]

Songhua also did wonderful work on "Automatic Generation of Artistic Chinese Calligraphy" [4]. They introduced a novel intelligent system that can generate new Chinese calligraphic artwork that meets certain aesthetic requirements automatically. Their system can derive parametric representations of existent calligraphic artwork from input images of calligraphy. They used a six-level hierarchical structure to learn the calligraphy (figure 3). A set of geometric constraints is proposed and incorporated into the system for rejecting unacceptable results. The combination of knowledge from various input sources creates a huge space for the intelligent system to explore and produce new calligraphy.

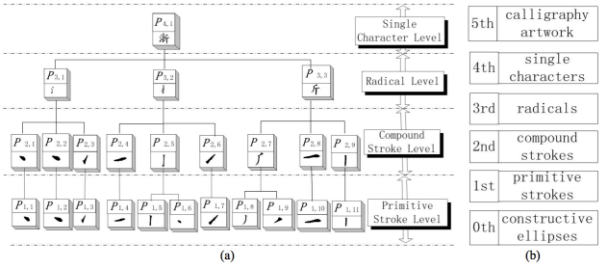


Figure 3

A lot of researches have been on style learning in the image processing area. Leon's paper [1]: A Neural Algorithm of Artistic Style is the most interesting and exciting one. It is the first paper addressing on the method to learn content and style of an image separately. The magic of mixing the content in picture with the style of painting comes from the design of loss functions.

$$G_{ij}^l = \sum_k F_{ik}^l F_{jk}^l.$$

Formula 1

For learning style, Leon uses gram matrix (formula 1) to describe the similarity between feature maps. F^l is just the feature map for each VGG net layer. By taking the sum of squares of two different G^ls from input and output. We get our loss function. For content learning, we directly use the sum of squares of the difference between two F^l calculated from input and output as the loss function. By applying back propagation and SGD, we can reduce our loss and get our desired output (figure 4).

Figure 4

III. Dataset Exploration

The font dataset we used is a collection of online fonts scraping from free fonts websites. After processing the raw images with cropping, rotating, scaling and converting them into bitmap we end up with 3.5 million letters in size 64*64. We categorize those letters into ~ 56,000 fonts and each individual font consists 62 letters from 'A' to 'Z', 'a' to 'z' and '0' to '9'. So we end up with a (56,443*62*64*64) tensor and Figure 5.1 may give you a sense

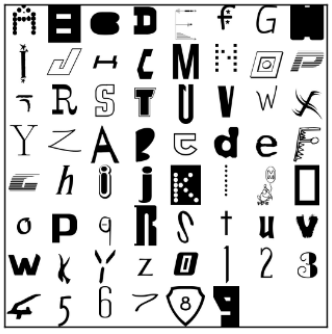


Figure 5.1: Letters randomly selected from the font dataset

where the number 62 comes from. Also in Figure 5.2 we present letter 'A' in the first 39 fonts in our dataset and a single font style with all 62 letters. It challenges our network in understanding huge number of fonts in such a diversity format.

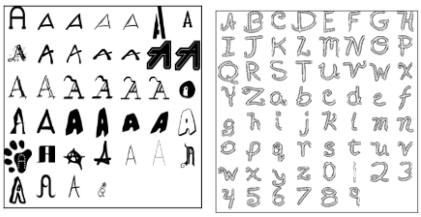


Figure 5.2: (Left) Letter 'A' in 39 different fonts (Right) 62 letters in one single font style

Moreover, as Figure 6 shows, it is very interesting that the average of all fonts is grey in its background. One possible explanation is that if we consider those letters in pixel level, thick fonts tend to have higher average value than the thin ones. Therefore, averaging all of them create a blurred letter with background in the about 0.5 in pixel which is grey in color. Realizing this help us in designing the loss function of network.

Figure 6: (Left) Average of all 56k fonts. (Right) Median of all fonts

IV. ALGORITHM

In this section, we will present the network specification in input and output first. Then description of the hidden layers of the network and training techniques we have adopted will be shown. Finally, discussion of training and testing set split will be provided.

From Figure 7, there are two inputs for the network: one-hot encoding vector of 56,000 fonts and one-hot encoding vector of 62 letters. In another word, we specify, at the beginning, the name of font style and letters to the network and we expect that our network can generate right format of the letter by itself. So, the output of the network is one 64*64 picture, a 4096 dimensional vector equivalently, and we compare it to the true format by calculating their mean absolute error. Recalling that in section 3 that the average of all fonts has a grey background. From our empirical result, we find out that using mean square error as loss function generates the same grey images as well. Therefore, we switch the loss function from L2 norm to L1 norm and practically it works well generating clear background in predicted images.

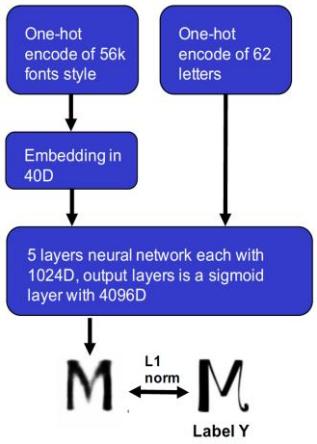


Figure 7: Specification of the neural network

One problem of taking one hot encoding vector as input is that there are too many fonts and the resulting vector is extremely sparse. Instead of dumping millions of sparse vectors into the network, we apply a transition matrix to project the sparse input into a 40-dimensional embedding space. We think this is very crucial to the succeed of our network because it not only speeds up the whole training process but also provides an opportunity in studying the relationship of various fonts in a much lower dimensional space. After the projection, we concatenate the 62 dimensional letter input with the 40 dimensional embedding into a 102 vectors as the final input layer. Then by stacking 4 fully connected layers each with 1024 dimensional width and leaky rectifier linear unit (alpha=0.01) as activation function, the network learns an abstract representation from the inputs. Finally, an output layer in width 4096 with sigmoid function as nonlinearity generates the pixel value between 0 (white) and 1 (black) in the output picture.

Referring to the optimization techniques of our neural network, we apply the following tricks:

- Add Gaussian noise ($\mu = 0$, $\sigma = 0.03$) to the 40-dimensional embedding
- Blur label images by a Gaussian filter as data augmentation
- Batch normalization on dense layers
- Dropout rate 0.25 on dense layers as regularization
- Strong L2 regularization lambda (10^{-7}) on all parameters
- Learning rate 1.0 and decay rate 0.3 if test performance does not improve in 3 epochs
- Batch size: 512

- Use adadelta instead of traditional SGD for faster training

Even with help of those techniques, the training time on each epoch is about 1 hour and the loss decreases very slowly, we will discuss how we speed up the training process in hardware in next section.

The last question left to the algorithm part is how we split the training and testing set. When designing our network and the training samples, we expect our network that by learning some letters in a certain font (for example 'A' and 'B' in Arial), it can infer what 'C' looks like in Arial. In order to do that the network may need to get a sense of what 'C' generally looks like by learning from some other fonts such as Times New Roman. We split our dataset, ~3.5 million photos in random with 10% as testing set and we believe this is the best approach c omparing to splitting the dataset based on either the font style or the type of letters.

V. SOFTWARE PACKAGE DESCRIPTION

To complete this project, the toolsets that we used are Lasagne [5], cuDNN [6], Tensorflow [7] and Aphche Spark. In this section, we will introduce how we utilize these tools to build network and conduct some studies of embedding space. The machine we used is an Ubuntu 14.04 system with one Tesla K80 GPU hired from Amazon AWS services.

Lasagne is a lightweight library to build and train neural networks in Theano. It provides some state-of-art implementations in various layers and optimization algorithms which enable users focusing on the testing performance in different architectures instead of getting

```
input_font_bottleneck_noised = lasagne.layers.GaussianNoiseLayer(input_font_bottleneck, sigma=font_noise)
network = lasagne.layers.ConcatLayer([input_font_bottleneck_noised, input_char_one_hot], name='input_concat')
for i in xrange(4):

    # Add batch norm layer and the dropout layer
    #network = batch_norm(lasagne.layers.DenseLayer(network, D, name='dense_%d' % i, nonlinearity=lasagne.nonlinearities.le
    network = lasagne.layers.DenseLayer(network, D, name='dense_%d' % i, nonlinearity=lasagne.nonlinearities.leaky_rectify)
    network = lasagne.layers.DropoutLayer(network,p=0.25)

network = lasagne.layers.DenseLayer(network, wh, nonlinearity=lasagne.nonlinearities.sigmoid, name='output_sigmoid')
```

Figure 8: sample code for building the network in Lasagne

frustrated for the implementation of those techniques. From Figure 8, we can import many layers, for instance, drop out layer, dense layer, Gaussian noise layer etc., simply from the library. If we build the all functionalities in Theano by ourselves, it would take us several days to accomplish and debugging them will create a lot of confusion. Lasagne frees us from those frustrations and enables us try more optimization techniques to achieve lower loss value and faster training time.

Since we mention the training time issue in previous section, we update the cuDNN to the latest version for the fastest implementation of those layers and tricks we talked beforehand. In short, The NVIDIA CUDA® Deep Neural Network library (cuDNN) is a GPU-accelerated library of primitives for deep neural networks. cuDNN provides highly tuned implementations for standard routines such as normalization, activation layers, forward and backward convolution and pooling. It is our first time to update cuDNN for a remote machine. We follow many great tutorials online to do that eventually. Fortunately, this update reduces the training time for one epoch from 1 hour to about 40 minutes.

After we have finished training the network, we use Google embedding projector [8] in Tensorflow to visualize our final embedding in 3D. Embedding projector is a tool for interactive visualization and interpretation of embeddings open sourced by Google. Currently it offers dimension reduction techniques including: principal component

Figure 9: User Interface for Google embedding projector

analysis (PCA) and t-Distributed Stochastic Neighbor Embedding (T-SNE) [9]. Figure 9 shows the UI for embedding projector and you are also welcome to check our demo video if you are interested.

The final tools we applied is Apache Spark, we use it to do some exploratory analysis including visualization of mean and median of all letters. Recalling that we have a huge (56,443*62*64*64) tensor storing on our laptop and it takes almost forever to calculate its mean in library such as numpy. We use Spark mainly to distribute that process. The main challenge is we store the data in a hdf5 file and currently Spark does not provide any official tool to read hdf5 format into rdd format. To solve that we write a small transition function to read our raw data into Spark rdd file system. For detailed information in how we do that, please check our read.py file on Github.

VI. Experiment Results

The final training time for the network converge is about 5 days, the loss starts from 0.15 and gradually decrease to about 0.05. In this section, we will first present sample output from the network and comparing it to the desired characters. Then analysis of the font embedding and visualization will be provided. Finally, we will show, by manipulating and interpolating the embedding space, 3 ways that machine can create its own font style.

Figure 10: sample output from testing set where the real character is on the left, the model output on the right.

To start with, we recreate real font characters with characters generated from the network (Figure 10). These are all characters drawn from the test set, so the network hasn't seen any of them during training. All we're telling the network is (a) what font it is (b) what character it is. The model has seen other characters of the same font during training, so what it does is to infer from those training examples to the unseen test examples. The network does a decent job at most of the characters, but gives up on some of the more difficult ones. For instance, characters with thin black lines are very hard to predict for the model, since if it renders the line just a few pixels to the side, that's twice the loss of just rendering whitespace.

By looking at the frequency plot of all fonts in the 40-dimensional embedding space, it ends up being roughly a multivariate normal (Figure 11). Also, by looking at Appendix 1, the 2 dimensional embedding space after reducing by T-SNE shows that the thin fonts are clustered in the middle and bottom left in the space while the thick fonts are clustered around those thin ones. In our Youtube video [10] it presents clearly, in 3-dimensional space, the thick

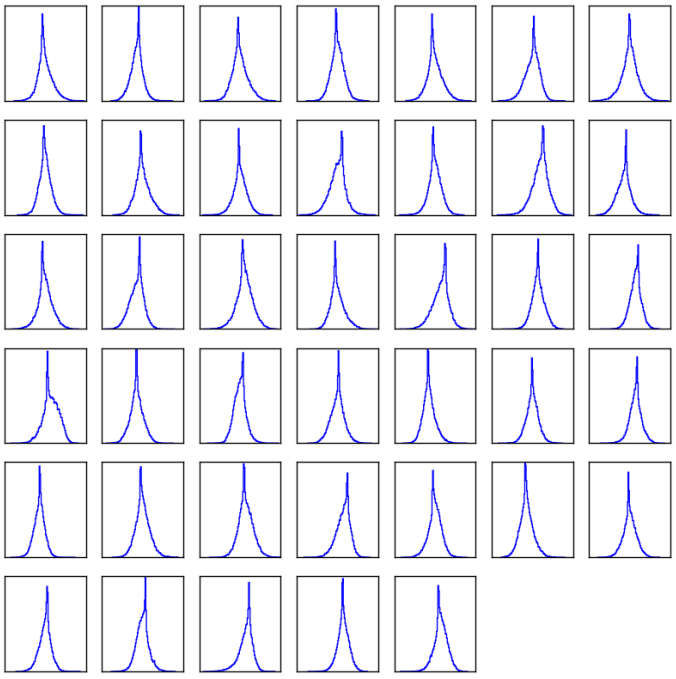


Figure 11： Visualization of 40D embedding of all 56k fonts, here horizontal axis: 56k fonts and vertical axis: frequency of values

fonts occupy about 75% of the ball while the rest are filled by thin ones.

In order to generate completely new fonts by machine itself, the first way we present here is interpolating between different fonts in continuous space. Since every font is a vector, we can create arbitrary font vectors and generate

Figure 12: generating new fonts by interpolating embedding space

output from it. In figure 12, we sample four fonts from the training set and put them in the corners of a square. Then by segmenting their distance in embedding space, our network can create mixture styles of four fonts at corner by simply calculating the weighted average in their embedding and printing out the new characters. An interesting thing here is

the model has learned that many fonts use upper case characters for the lower case range — the network actually interpolates between 'H' and 'h' seamlessly. Figure 13 is an example demonstrating that.

Figure 13: the network interpolates between 'H' and 'h' seamlessly

Second approach to generate new fonts is quite easy to understand, our network can pick a font vector randomly and generate new fonts by some perturbations, for example adding small uniform noise.

The third method to create new fonts is by modeling the distribution of font vectors as a multivariate normal, we can sample random vectors from it and look at the fonts they generate. Figure 14 is a new fonts created by sampling from multivariate normal distribution. There is still some room for improvement as you may see the edges of fonts are blurred and we believe by further reducing the loss of network, we would obtain fonts with better quality by this approach.

Figure 14: new fonts created by random sampling from multivariate normal distribution

VII. Conclusion

As a group of only two students, I thought two biggest constraints we had during this project were resources and time. When we cut off the training program due to time limitation, the loss for the network kept decreasing slowly, which means if we have more time, we would end up with fonts in better quality. Also this was our first time training such a large dataset, it took us a lot effort setting up Amazon AWS services and tried numerous things from writing better code to hardware upgrade to make sure we would obtain a reasonable result on time. We were proud that our neural network finally generated the predicted characters in a pretty good shape (for most of them) and we figured out how to read hdf5 file into Spark. Also we tried a new tool: embedding projector in visualizing high dimensional data. All those experiences will support us and push ourselves to the next level in big data research.

For the next step, we will explore generative adversarial models [11], which seem better at generating pictures. Also since we have mentioned in section 2, we can explore new architecture including convolutional and deconvolutional layers instead of fully connected layers and design completely new loss function as introduced from Leon's paper [1]. Moreover, we have discussed at the beginning of our project, we are very interested in applying this technique on Chinese character. As an extension to this project we will continue collecting Chinese fonts from the web and we look forward to present our network on Chinese character in the future.

Acknowledgment

We would like to thank Prof. Lin and all the TAs for offering us such a wonderful semester. Also we would like to thank Prof. Tian Zheng for her advices and finally Erik Bernhardsson in collecting such a great dataset.

APPENDIX

REFERENCES

[1] L. A. Gatys, A. S. Ecker, and M. Bethge. A neural algorithm of artistic style. arXiv preprint arXiv:1508.06576, 2015.

[2] Neill D.F. Campbell and Jan Kautz1. Learning a Manifold of Fonts. Link:http://vecg.cs.ucl.ac.uk/Projects/projects_fonts/papers/siggraph14_learning_fonts.pdf

[3] http://vecg.cs.ucl.ac.uk/Projects/projects_fonts/projects_fonts.html

[4] Songhua Xu, Francis C.M. Lau, Kwok-Wai Cheung and Yunhe Pan. Automatic Generation of Artistic Chinese Calligraphy. Link: https://www.aaai.org/Papers/IAAI/2004/IAAI04-024.pdf

[5] Introduction page of Lasagne. https://lasagne.readthedocs.io/en/latest/

[6] Developer page of cuDNN. https://developer.nvidia.com/cudnn

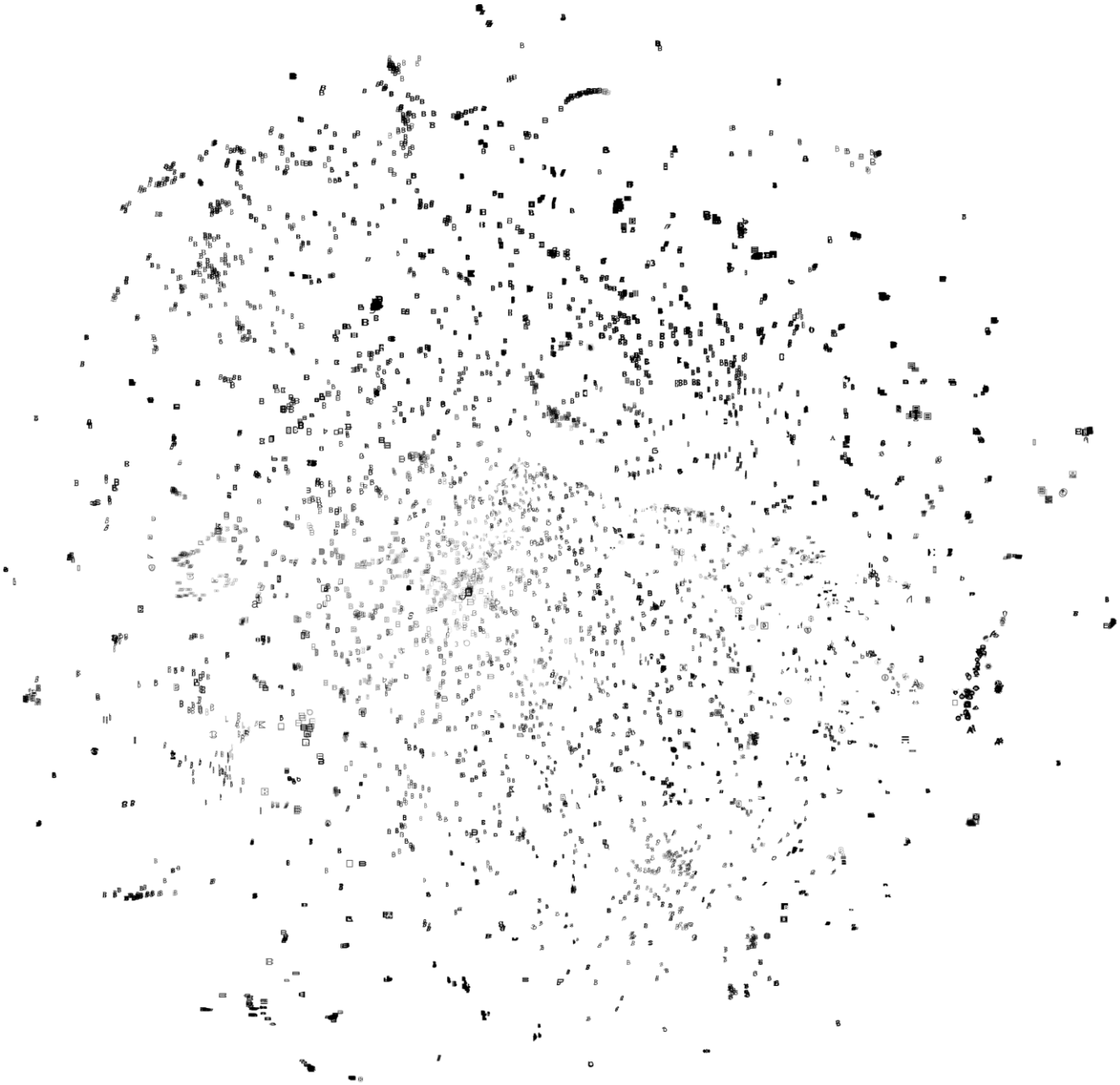
[7] Homepage of Tensorflow. https://www.tensorflow.org/

[8] Homepage of Embedding Projector. http://projector.tensorflow.org/

[9] Laurens van der Maaten: Introduction of T-SNE. https://lvdmaaten.github.io/tsne/

[10] Video URL https://www.youtube.com/watch?v=UYrIfKr41ts.

[11] Ian J.Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio. Gnerative Adversial Networks :2014. https://arxiv.org/abs/1406.2661



Appendix 1: Visualization of embedding by T-SNE in 2D