Chinese Calligraphy Font Classification and Transformation

Li Deng Liyi Wang Zhaolin Ren

[a] *SUID: dengl11 liyiw rzl*

Abstract

This project explores Chinese character font classification and transformation, which are the most important two steps in reconstructing weathered Chinese characters. For font classification, *SIFT* is first used to capture font features, and neural network is built in *Keras* with almost perfect results achieved. For font transformation, a convolutional neural network is implemented in *Tensorflow*, for which different models are compared and discussed. Expected results are generated from transformation, and effectiveness of neural network in font classification and transformation is verified.

1. Introduction

As an invaluable part of ancient Chinese culture, Chinese calligraphy has had a long and brilliant history. However, many ancient Chinese calligraphy works were carved on stone tablets. After years of weathering, many of them have become damaged and incomplete. Inspired by machine learning, we thought to ourselves: what if we were able to make computers learn the font styles of these ancient works, and ultimately render the incomplete characters in the original font?

This project aims to make a contribution towards that goal, for which two steps are essential. First, font types need to be identified, since these form the target font style for reconstruction. The next step, creating the same character in the target style, is more challenging. Instead of starting from scratch, we convert the problem to a transformation from a base font into a target font. Below is an example: we have trained a model to learn some characters in *Kai* font in order to reconstruct a new character which is not in our training set. We take the same character in the base font, and transform it into the font of *Kai*.

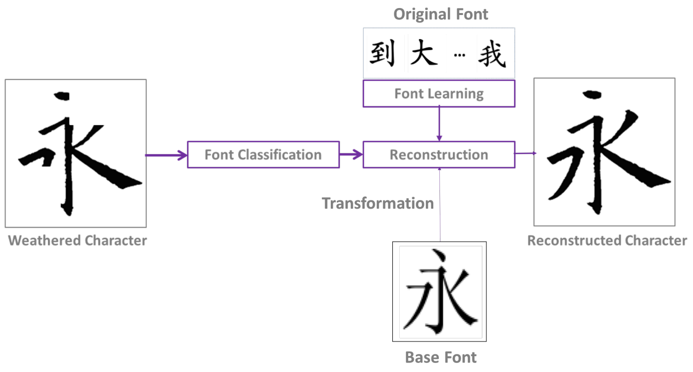


Figure 1 Classification and Transformation in Reconstruction Workflow

2. Related Work and Challenges

There has been some but not significant literature dedicated to the classification of Chinese font styles. Lin [1] makes use of *GIST* image features to classify Chinese font types, while Mao [2] uses both *GIST* and *SIFT* features to perform the same classification task. We also found a *CS231N* project that made use of convolutional neural networks to classify different Chinese font styles [3]. Style transfer of different font types is a relatively less-studied topic. The best source we found on this was Baluja [4], who implemented a deep neural network model, using Fully-Connected Layers, to perform font style transfer learning of English fonts. However, we found no existing papers that looked into style transfer learning across different Chinese font styles.

There are mainly two challenges to this project:

- **Difficulty in obtaining training data**: As actual incomplete calligraphy characters on stone carvings are hard to obtain, we adopt a simplified approach: first, we use standard font file(*.ttf* or *.otf*) to generate images of characters in different fonts, and then train the network to transform the base font to the target font.

- **Extraction of features relevant to font style**: Chinese characters are complex in shape and variance. For a given image of one character, how might we extract features that are relevant to its font style, instead of other attributes, like density of strokes? The first idea is to use *kNN* for feature filtering, and the next idea is to let the neural network automatically select the features by supervised learning; both these approaches are discussed in detail in the Methodology section.

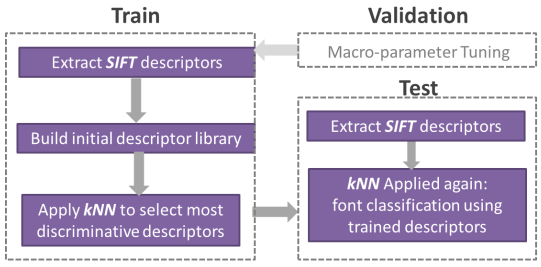
3. Training Data

We focused on five Chinese fonts: *Kai*, *Li*, *Song*, *Xing*, and *Zhuan*, for which font files(*.ttf*) are obtained. Each of them contains the same 3011 characters. Based on the font file, for our *Matlab* SIFT implementation, images are generated for each character. Meanwhile, for running the neural network in *python*, to reduce the overhead of reading images, font files were converted directly to *NumPy* file, which contains a single 80×80 pixel grayscale bitmap for each character (so total dimension is $[3011 \times 80 \times 80]$). For font transformation, considering that the *maxpooling* step before the output layer halves the image dimensions, we doubled the dimension for each character in the source font to 160×160.

4. Methodology

4.1. Feature Filtering Based on SIFT-kNN

As a baseline, a *k-Nearest Neighbor (kNN)* classification model is first used. As a feature extraction model in the *VLFeat* library, *SIFT* is employed to extract the important local descriptors (features) from 3011 images of characters with their font type labeled. Out of these descriptors, a further filter is then applied to select only the descriptors that were best able to match the font label. In the test step, *SIFT* descriptors are first extracted from the new character to be compared with filtered descriptor library constructed in the train step. Then for each character, the *kNN* approach is applied again to identify which font most of the SIFT descriptors for this specific character correspond to. Then, the font type having the most descriptors is predicted for this character.



Train
Validation
Macro-parameter Tuning
Extract SIFT descriptors
Build initial descriptor library
Apply kNN to select most discriminative descriptors
Test
Extract SIFT descriptors
kNN Applied again: font classification using trained descriptors

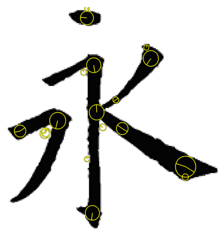


Figure 2&3 *SIFT-kNN Algorithm, SIFT Descriptors*

In the example character shown above, descriptors produced by running *SIFT* are displayed as yellow circles. But only a few of them describe font styles. Running the algorithm outlined above extracts and builds the font descriptor library for classification.

4.2. CNN-based Classification

With satisfactory but not perfect results obtained by *SIFT*, we turn to *Convolutional Neural Networks (CNNs)* so that features can be automatically selected by the network. For each font, a training set of 450 characters and a test set of 50 characters are used. 32 filters of $[5 \times 5]$ pixels are used with stride of 1 in 1 convolution layer. Activation weights are normalized to reduce overfitting. *Re-LU* introduces non-linearity into the system. Before the output, the following three parts are included:

- **Max-pooling**. Here, we take each 2 by 2 pool and and replace it with a single value corresponding to the largest value in the pool, with a stride of 2. This helps to reduce the dimensionality and number of parameters of the network, hence reducing the risk of overfitting.

- **Dropout**. We adopted a dropout rate of 0.9, which means that we kept each neuron active with probability 0.9. Dropout has been shown to be an effective anti-overfitting technique [8].

- **Softmax**. Softmax layer outputs a $[5 \times 1]$ vector indicating the probabilities of the character belonging to a certain font.

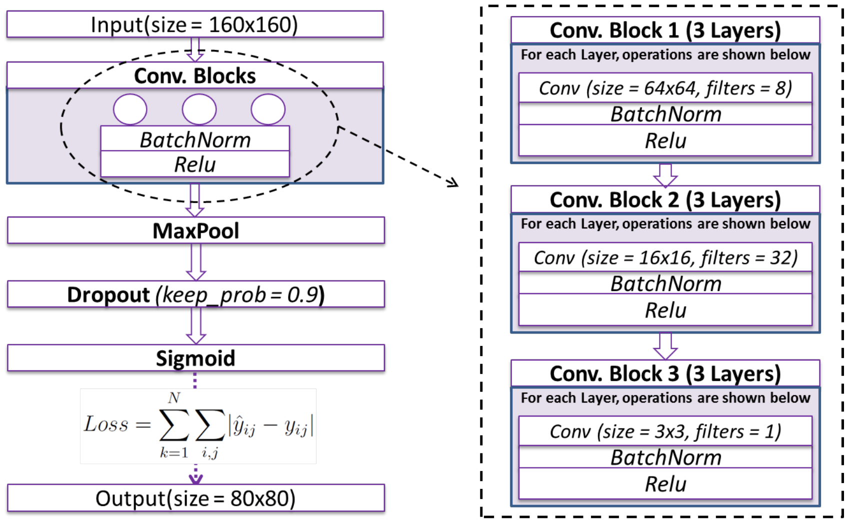
4.3. CNN-based Font Style Transfer

Two fonts are used for transformation, one source font and one target font. Given an image from the source font, the output is expected to resemble the image of the same character in the target font as closely as possible. Hence, the loss function is defined to be the sum of absolute pixel-wise differences:

$$Loss = \sum_{k=1}^{N} \sum_{i,j} |(\hat{y_k})_{i,j} - (y_k)_{i,j}|$$

where N is the batch-size used at each training step, $\hat{y_k}$ is the output pixel matrix at the end of the CNN process we obtained from x_k, the pixel matrix of a character in the source font style, and y_k is the pixel matrix corresponding to the same character in the target font style. The reason for choosing to minimize the L^1 loss rather than the L^2 loss is that L^2 loss-minimizers tend to perform poorly on image generative tasks when evaluated against the perceptions of human observers [10]. Our architecture can be seen below.

Parameterized by filter size, three convolution blocks are used, each of which contains a certain number of layers $nLayer$. For $nLayer = 3$, the network architecture is shown below:



Input(size = 160x160)
Conv. Blocks
BatchNorm
Relu
MaxPool
Dropout (keep_prob = 0.9)
Sigmoid
$Loss = \sum_{k=1}^{N} \sum_{i,j} |\hat{y}_{ij} - y_{ij}|$
Output(size = 80x80)
Conv. Block 1 (3 Layers)
For each Layer, operations are shown below
Conv (size = 64x64, filters = 8)
BatchNorm
Relu
Conv. Block 2 (3 Layers)
For each Layer, operations are shown below
Conv (size = 16x16, filters = 32)
BatchNorm
Relu
Conv. Block 3 (3 Layers)
For each Layer, operations are shown below
Conv (size = 3x3, filters = 1)
BatchNorm
Relu

Figure 4 *Convolution Neural Network Structure*

Without repeating our earlier discussion of the CNN architecture, we will focus on the changes we made from the CNN classification model.

- Given the anticipated challenge of a style-transfer learning task, we felt that it was important to try to capture both large-scale and small-scale features. Correspondingly, we included three convolutional blocks: first, eight 64 by 64 pixel filters, followed by thirty-two 16 by 16 size filters, and then sixty-four 3 by 3 filters. We tried different number of convolutional layers within each block, ranging from 1 to 3.

- A sigmoid layer follows the dropout layer, where we took the output matrix following the dropout step and performed an element-wise sigmoid computation of the matrix. This helps to make the pixels closer to either being completely black or white. It is observed that generated images suffer from heavy background noise without this sigmoid layer.

- **Batch Size and Random Permutation**: At each train batch step, a batch size of characters are randomly selected from training set. A greater batch size helps the training loss to decrease faster, but requires more runtime. A batch size of 16 is finally used to keep the balance.

5. Results and Analysis

5.1. k-Nearest Neighbor SIFT-based Classification

A SIFT descriptor library is obtained for 1000 characters in each font. As can be seen in the confusion matrix shown on the right, the prediction accuracy is satisfactory for *Song*, *Li*, *Kai* and *Zhuan* font, all of these are above

95%. But for the font of *Xing*, which is more cursive, the accuracy is only 91%, where quite a number of the characters are wrongly predicted as *Kai* font. In tuning the parameters of the *kNN* algorithm, we experimented with values ranging from 1 to 20 for the k_1 and k_2 parameters, and finally found that setting both k_1 and k_2 to be 3 gave us the best accuracy.

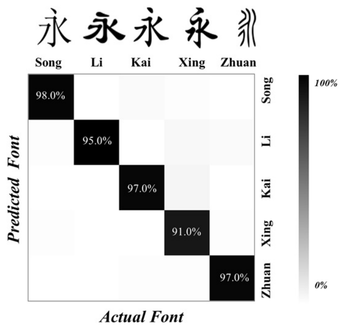


Figure 5 *Confusion Matrix of Classification by SIFT*

5.2. *CNN-based Classification*

CNN-based classification is implemented with *Keras* package in Python. Our training and testing results are shown in the chart below. As can been seen, using 2250 training characters, the model converged swiftly and achieved a perfect classification outcome on the testing set, which contained 250 characters. This excellent classification performance, obtained from having just a single convolutional layer, illustrates the effectiveness of CNNs in image classification tasks.

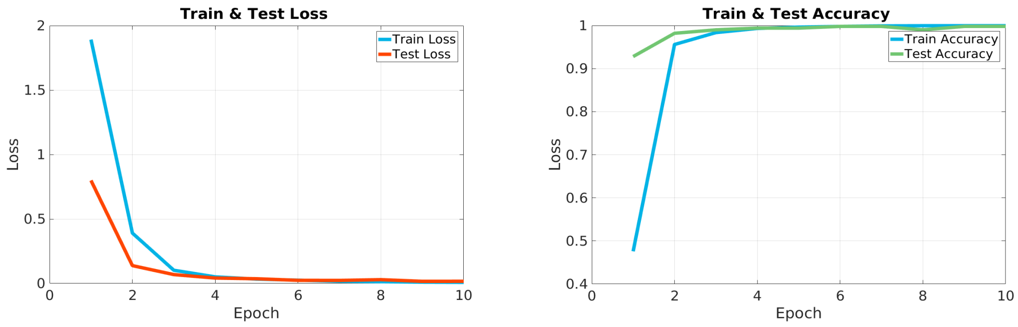


Figure 6&7 *Loss, Accuracy Diagram*

5.3. *Comparison between two classification approaches*

After comparing the results of two approaches above, it is clear that CNN-based classification yielded more accurate predictions, despite having just half of the training dataset for the SIFT approach. The reason is that SIFT descriptors describes more features on local details of an image, while CNN is able to describes both local and global features.

5.4. *CNN-based Font Style Transfer*

We obtained reasonably good results for our transfer learning process, where we found that the training loss declined steadily. With more layers added, a smaller final training loss is achieved, as the plot shows below:

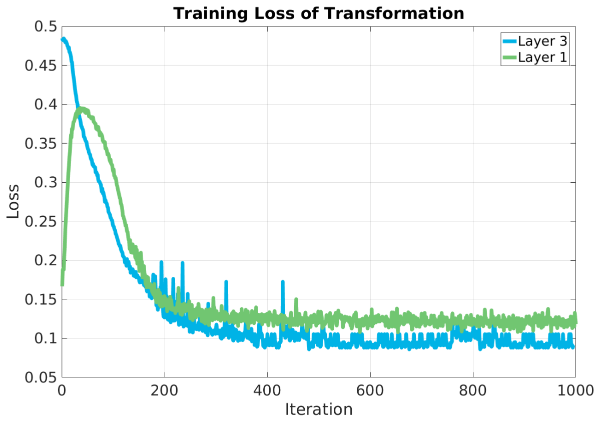
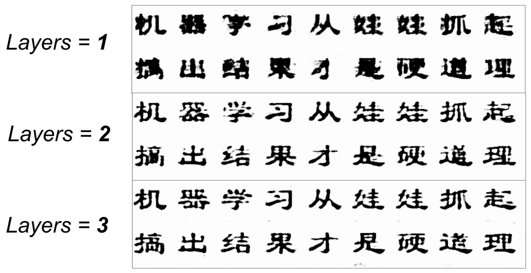


Figure 8 *Train Loss-Iteration Diagram*

Convergence of the training model verifies the correctness of training, and generated images on test characters are then used to evaluate the effectiveness of training. By comparing multiple sets of outputs by using different parameters, the main observations are outlined below:

- **Number of layers**: 1 to 3 layers are experimented for each block, and not surprisingly, more layers produce clearer images;



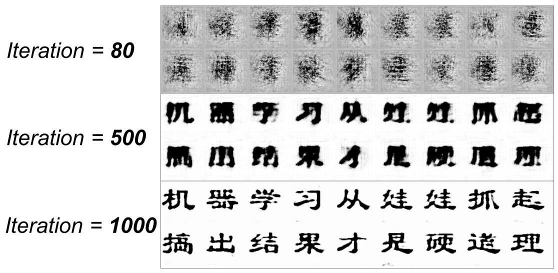
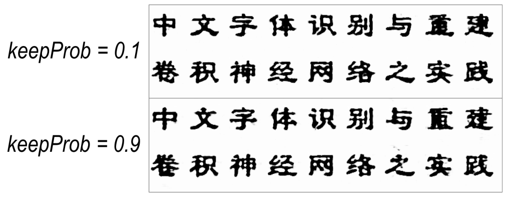


Figure 9 Comparison of Number of Layers   Figure 10 Transformation Process

- **Loss Optimizer**: A standard batch gradient descent optimizer was first tried but converged rather slowly. The *RMSProp* optimizer, which makes use of adaptive learning rates, is observed to converge much faster.
- **Keep Probability in Training**: As Figure 11 below shows, a smaller dropout rate produces sharper characters with less noise.



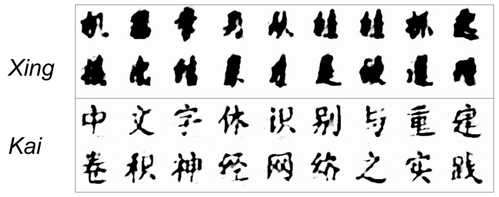


Figure 11 Comparison of *Keep Probability*   Figure 12 Comparison of Target Fonts

Transformations above are all from *Song* to *Li*. We also tried using more complex target font styles with less success. Shown in Figure 12 above is the output for *Xing*, whose strokes are much less distinguishable than the more regular font of *Kai*.

6. Conclusion

Aiming at the ultimate goal of Chinese calligraphy character reconstruction, this project explores font classification and transformation by *SIFT* and *Convolutional Neural Network*. Almost perfect results are obtained for classification, and satisfactory performance is observed for transformation between fonts with regular shapes.

Future work may extend to:

- Improvement of transformation to more complex font: only blurry characters are produced in this project so far for complex fonts like *Xing*. Since real calligraphy can be even more complex, it is necessary to enhance the performance of the network by using either more sophisticated structure or more training data. As a generative model, *Generative Adverserial Networks (GANs)* can also be explored for the same purpose.
- Reconstruction from real calligraphy characters: this project approaches the challenging problem by a simplified model. More needs to be done for reconstructing real calligraphy characters, like background noise deduction, and character segmentation from a whole page of characters.

7. Github & Demo

This project is hosted at github. The reader can view our demo at the project website

8. References

[1] Lin,Yuan. *Research and Application of Chinese Calligraphic Character Recognition.* Zhejiang University, Zhejiang, China, 2014.

[2] Mao, Tianjiao. *Calligraphy Writing Style Recognition Technology.* Zhejiang University, Zhejiang, China, 2014.

[3] Boqi, Li. *Convolution Neural Network for Traditional Chinese Calligraphy Recognition* Stanford University, CS231N Class Project, 2016.

[4] Baluja, Shumeet. *Learning Typographic Style* Google, Inc, 2016.

[5] Jason Brownlee. *Develop Your First Neural Network in Python With Keras Step-By-Step.* http://machinelearningmastery.com/tutorial-first-neural-network-python-keras/.

[6] VLFeat.org. *SIFT detector and descriptor.* http://www.vlfeat.org/overview/sift.html.

[7] Stanford CS 231N. *CS 231N Class Notes* http://http://cs231n.github.io/

[8] Geoffrey E. Hinton and Nitish Srivastava and Alex Krizhevsky and Ilya Sutskever and Ruslan Salakhutdinov. *Improving neural networks by preventing co-adaptation of feature detectors* http://arxiv.org/abs/1207.0580

[9] Tian, Yuchen. *Rewrite: Neural Style Transfer for Chinese Characters* Github Repository, 2016. https://github.com/kaonashi-tyc

[10] Hang Zhao, Orazio Gallo, Iuri Frosio, and Jan Kautz *Loss Functions for Neural Networks for Image Processing* 2015. https://arxiv.org/abs/1511.08861