Semester project: Chinese type design

Maxime Schoemans
École polytechnique de Lausanne, Lausanne, CH
maxime.schoemans@epfl.ch

June 7, 2019

Abstract

This project describes a method to reduce the tedious manual work of Chinese type designers when designing a new font, by reducing the number of characters that have to be manually drawn from a few thousand to a few hundred. As a first step, a method for combining these characters is described, which results in the creation of a rough version of all the images from the complete character set. Finally, a Unet, trained in character-to-character translation, is used to refine the previously created images. The final version of the project is able to create a character set of around 3000 characters starting from a set of around 300 manually drawn characters. The best version of the character-to-character translation network achieves an IoU on the validation set of 82.8%. Eventually we conclude that a comparable method can be used to create decent output characters and that the method should be improved before it can be used as a real way of generating characters for new fonts.

1 Introduction

Chinese type design has always been and remains a very manual job requiring a lot of character design to produce a complete set of characters in a new font. This work has already been simplified by graphics editors but remains a huge task due to the large number of existing Chinese characters. Previous work has been done to try to simplify [1] or automate the creation of new fonts, by either using the decomposition of Chinese characters [2] or trying to differentiate character embedding from style embedding and using style transfer to create new fonts [3, 4]. This work tries to examine if it is possible to automate the creation of a complete character set in a new font starting from a small set of manually drawn characters. For this, both the character decomposition method and an adaptation on the network used in [3] are combined to makes this happen.

2 Project Structure

The automatic creation of Chinese characters in a new font is separated into two main parts. Firstly, a small set of characters ($\sim$ 300 characters, called the 'basic characters' from now on) are combined together using specific layouts to create the complete Chinese character set, as explained in Section 3. This character set will be called the 'rough character set' since the characters are created using simple layouts and still need to be improved. In a second step, the rough characters are put through a neural network that is trained to 'refine' the characters and that is described in Section 4. The result of this step is thus a 'refined character set' in the new font, which corresponds to the result of the project. To be able to use this method, the base characters first have to all be manually drawn in the new style, before they can be used as input of the first step.

3 Rough character set

The first step of the project consists of combining a small set of base characters using different layouts to create a larger set of rough characters. To this end, a 'character definition' has been added to each character of the final set.

To create a new font, a minimum character set containing the 6-7000 most frequently used characters has to be defined. This can then later be expanded to ∼20.000 for a larger character set and even to ∼70.000 for the complete set if needed. However, since adding a definition to every character in the final set takes a non-negligible amount of time, a smaller character set of around 3000 character has been used for this project.

3.1 Character definition

Most Chinese characters are combinations of two or more simpler characters and they can thus be defined as being composed of these characters, using a certain layout. A good example of this can be seen in Figure 1.

亢 亠 几

Figure 1: Character with its components

Every character of the final set has thus a 'character definition', describing what its components are and which layout to use to combine them to recreate the corresponding character. Two different layout formats can be used: a general layout format, described in Section 3.2, or a precise layout definition, described in Section 3.3.

A database has been created to store these definitions. As said previously, this database is not complete and contains only around 3000 characters at the moment, but more definitions can be added to the database easily. Below is an example of a character definition without a precise layout definition.

- Character: 亢
- Unicode: 4ea2
- Layout id: 2
- Components: 亠, 几
- Component unicodes: 4ea0, 51e0
- Precise definition: None

Figure 2: Example of a character definition

3.2 Layouts

To combine multiple components together, we first define a set of 36 general layouts. The complete list of layouts can be found in Appendix A. The process of creating an image of a character using its component and a layout is then simply to take the images of its components, resize them accordingly and paste them together as defined in the layout.

The issue with using this method is that the correct proportions and the width of the strokes are not respected. This is due to the fact that the resizing changes the width of the strokes too much and that the layouts are too general to handle the different proportions in the characters. A slight improvement was thus made by verifying if the component can fit entirely inside the target layout before resizing its image. If this is the case, the component does not need to be resized at all but only moved, and this improves the width of the strokes in the created images. The difference between the results of the two different methods can be seen in Figure 3.

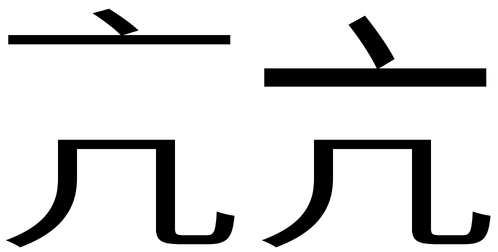


Figure 3: Character images created using a general layout. Left is the initial method and right is the improved method.

The base characters are the characters that cannot be decomposed further into components. These characters are defined as having one component, themselves, and layout number 0. An image of every base character has thus to be provided to be able to create the rest of the character set. When all characters of the set have a definition and the images for all base characters are provided, it is then possible to automatically produce the images for all characters in the complete set.

3.3 Precise definitions

To handle the issue of the proportions, we used a different approach. For this approach, a precise layout definition has to be defined for each character, where each of the components of the character is given a rectangle to fill in the target image. This technique creates much better results than the first one and is also more costly to implement and is thus not implemented for all characters.

To produce these precise definitions, two alternatives are possible: a manual approach, which consists of finding the bounding box of each of the components inside the image of the characters using a graphics editor or an automatic approach described below. The manual way is more time-consuming but can be applied to all characters. The automatic process, however, can only be used if the different components are separated by at least a stroke of white pixels in the image, which means that a manual preprocessing step has to be done to each character to determine if the technique can be used or not for that specific character. The reason behind this will be explained later.

3.3.1 Brute-force implementation

The first technique for automatic layout definition uses brute-force and consists of trying a lot of different sizes and positions for the components and choosing the one that overlaps the best with the target image. This method did not lead to the desired result and was very computationally intensive, so the idea was put aside.

3.3.2 Patch-based implementation

The second implementation relies on extracting black patches from the target image and comparing the different combinations of patches with the components. A black patch is a set of black pixels that are all connected together. Figure 4 shows an image of a character with all of its patches surrounded by red boxes.

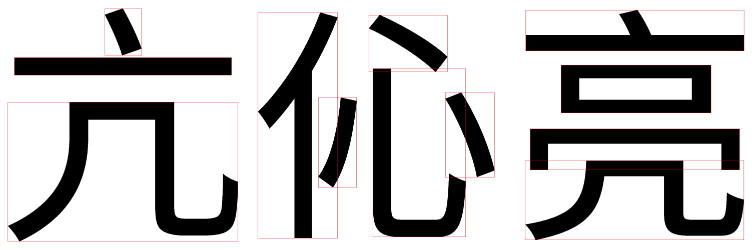


Figure 4: Characters with their black patches surrounded by red boxes

A flood fill algorithm is used to assign each black pixel in the image to a specific patch. Each patch is defined by its bounding box, surrounding all the pixels of the patch. When all the patches are computed, every combination of patches is tested for the presence of a component. Since a combination of multiple patches is also defined by a box surrounding all the pixels of the different patches, it can be thought of as a special kind of patch. To find a component in the image, the component is resized and overlapped with the image at the positions defined by the boxes of the different patches. The mean square error between the image and the component inside that bounding box is then used as a loss value, and the box producing the lowest loss value is assumed to contain said component. This method produces significantly better results than the first version and is thus used for this project, with some minor modifications explained in the next section.

3.3.3 Additional tests and adjustments

A first issue with the described implementation is the fact that some characters contain multiple times the same component. In that case, the previous method would return the same bounding box for all instances of the component, which is problematic. An easy fix for some of these characters is just to count the number of patches in the image and to assign each of them to a component if the number of patches found is the same as the number of components. However, since some components are composed of multiple patches, this does not work for all characters.

A particularity of the previous method is that the loss is computed between the resized component and the region of the image of the character defined by the bounding box of the patches. This means that it is possible for a different patch to be inside the compared region if the boxes of multiple patches overlap. This results in the loss being not entirely correct in some cases and caused issues for a few characters. To solve this problem, the loss computation was done between the resized component and a newly created image containing only the necessary patches of the target image. This method was able to solve the issue for some characters but created new issues in others, so the idea was finally abandoned. The comparison between the two methods can be seen in Figure 5.

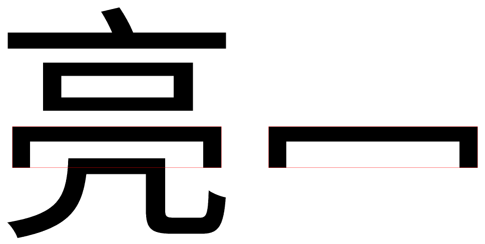


Figure 5: Example of images used when comparing with a component in the first (left) and second (right) method

Figure 6 shows the benefit of using a precise layout over a general layout. The precise layout definition clearly produces better results than the general layout.

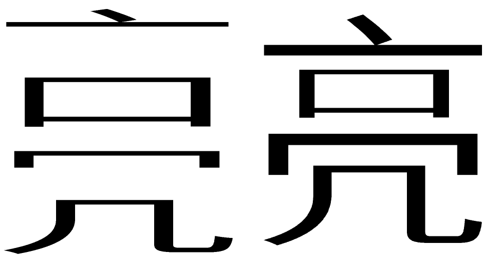


Figure 6: Comparison between using a general layout (left) and using a precise layout definition (right)

Since the goal is to use this precise definition for all fonts, the hope is that it can generalize well. An example of the usage of the same precise definition for the same characters in multiple fonts can be seen in Figure 7 and shows that it is indeed generalizing quite well to different fonts.

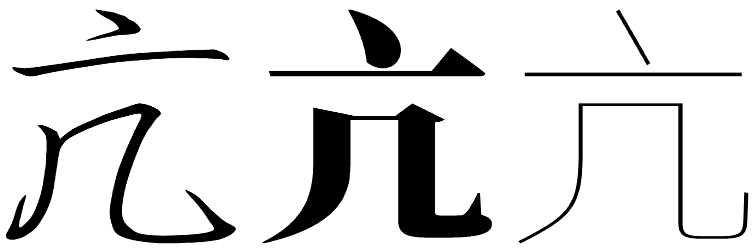


Figure 7: Use of the same precise definition in multiple fonts

4 Refined character set

In the first section of the project, the images for all the characters of the final set are automatically produced in the newly created font. Since these characters are produced using imperfect definitions, they do not look exactly like real characters and have thus to be refined further. To this end, a neural network has been trained to produce refined characters when given rough characters as input. The next sections will present multiple architectures possible for the network and will discuss the results of these different implementations.

4.1 Architectures

The two main architectures tested are a Unet for image-to-image translation and a GAN, composed out of the same Unet, combined with a discriminator. These architectures are slightly modified versions of the GAN architecture used in [3]. The unet is 8 layers deep, has 64 filters in the first layer, doubled every layer, and uses batch normalization and a ReLu as the activation function. The full architecture of the Unet can be found in Appendix B, together with the architecture of the discriminator used in the GAN.

4.2 Loss functions

Multiple loss functions have been added to the model, each with a corresponding weight that can be modified during the tests. The total loss for the Unet consists thus of a loss L_{l1} , defined as the mean absolute error between the output and target image, a loss L_{edge}, defined as the mean square error between the edges of the output and target image, and a loss L_{const}, defined as the mean square error between the encoded input and output image.

$$L_{Total} = w_{l1} * L_{l1} + w_{edge} * L_{edge} + w_{cons} * L_{const}$$

Of course, when training the GAN, the discriminator loss is also optimized.

4.2.1 Edge loss

Chinese characters are traditionally composed of strokes, meaning that the edges of a character are relatively smooth for most fonts. When training a Gan or Unet with only the L_{l1} loss, this particularity is not reproduced in the output. To attempt to solve this issue, we added an edge loss, which tries to penalize incorrect edges in the output images. Two different versions of the edge loss have been implemented and are described in the next sections.

Version 1

The first step to compute the loss is to identify the edges in the output image. To do this, a convolutional layer with a special 3x3 kernel is used. First, the maximum and minimum values inside the filter are computed. An edge is then detected if the difference between these two values is larger than a threshold.

$$\begin{bmatrix} 0 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix} \rightarrow edge \qquad \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \rightarrow not\ edge$$

Figure 8: Example of 3x3 activation kernels classified as edges or not

The initial goal of the edge loss was to penalize edges far away from the target edges more than edges that are close to the target edges. To this end, a distance map of the target image is precomputed and used to calculate the loss. This distance map has a value for each pixel that is proportional to the distance to the closest edge in the target image. After having detected the edge in the output image, the loss is then computed by doing the element-wise multiplication between the matrix with the edges of the output and the distance map. The final value for L_{edge} is the mean of the non-zero values of the resulting matrix.

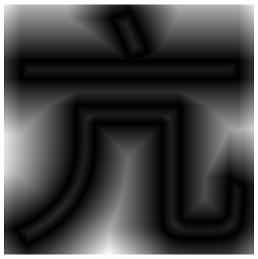


Figure 9: Distance map of a character, with the grayscale color corresponding to the value of the distance to the closest edge

The combination of the edge detection layer and the loss computation produces loss values that correspond to what was expected, meaning that the values are higher when the edges are far from the target edges and lower when the edges are close to the target edges. Unfortunately, this loss, together with the edge detection layer, is not fully differentiable and is thus not applicable in the network. Both ideas were thus abandoned and replaced by the second version.

Version 2

This second version of the edge loss corresponds to the mean square error between the edges of the output and target image. This loss is a special case of the edge agreement loss described in [5]. Based on the same paper, we create the edge detection layer by using the two Sobel filters [6] to identify the horizontal and vertical edges in the images. This version is used as the current implementation for the L_{edge} loss. The effect of this loss on the accuracy of the network is reported in Section 6.2.

4.2.2 Constant loss

The L_{const} loss is a loss that was implemented in [3] and it supposedly has a positive effect on the convergence during the training. This loss is described as being the mean square error of the encoded input and output images. Different networks are tested with and without this loss to see if it also has a positive effect on the training for this project and the results can be seen in Section 6.2.

5 Implementation details

The first part of the project is done using an image resolution of 1000x1000 pixels for all characters but for the second part the images are downscaled to 256x256 pixels, to be usable in the network. Both network architectures are trained on a single GPU using an NVIDIA GeForce GTX TITAN Z during 200 epochs, with batches of 16 images. For the optimization, an Adam optimizer is used with an initial learning rate of 0.001, halved every 10 steps down to a minimum of 0.0002. The character set consists of 2325 character in a single font, where 90% is used as the training set and 10% is used as the validation set.

6 Experiments

For the second part of the project, multiple experiments have been done to compare the different network architectures, the advantages of the different losses and the effect of image augmentation on the overfitting. For these tests, the intersection over union (IoU) is used as a metric to compare the results.

6.1 Influence of the architecture

The first experiment compares the results of training a Unet and a GAN for both 200 epoch using only the L_{l1} loss, with and without image augmentation. The image augmentation is done by flipping the images horizontally and vertically or by rotating them by 180 degrees. The results of this experiment can be seen in Figure 10 and Table 1.

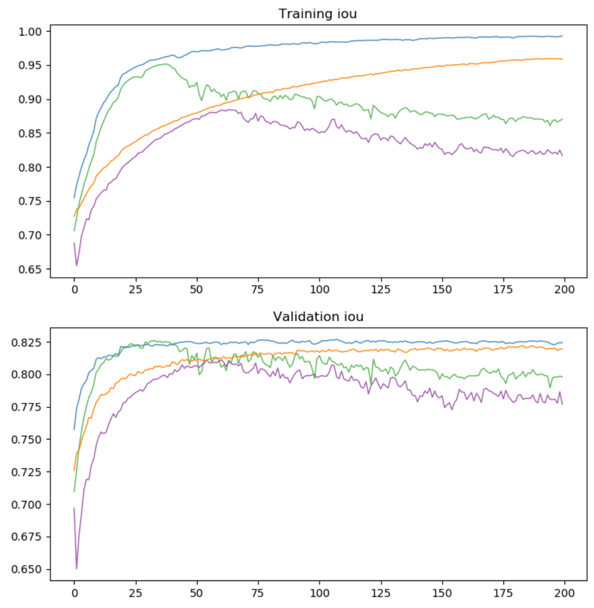


Figure 10: Effects of the architecture and image augmentation. See Table 1 for the legend

Model	Image augment.	Max val. IoU	Color
Unet	No	0.827	blue
GAN	No	0.826	green
Unet	Yes	0.822	orange
GAN	Yes	0.811	purple

Table 1: Results of experiment 1

When comparing only the Unet and the GAN, we can see that they achieve similar results on the validation set but that the GAN quickly reduces its accuracy after 30-60 epochs. The image augmentation slows down the convergence and does not seem to improve the results on the validation set, at least not when training for the same amount of time. The best and fastest network thus seems to be a simple Unet without image augmentation. This is the setup that is used for the next experiment.

6.2 Effects of the different losses

This second experiment examines the effects of the different losses on the accuracy of the network on the validation set. As said in the previous section, we use a Unet without image augmentation for this experiment. Four runs have been done, each having different values for the weights of the losses. The four runs with their corresponding weights can be seen in Table 2, together with the best value for the validation IoU.

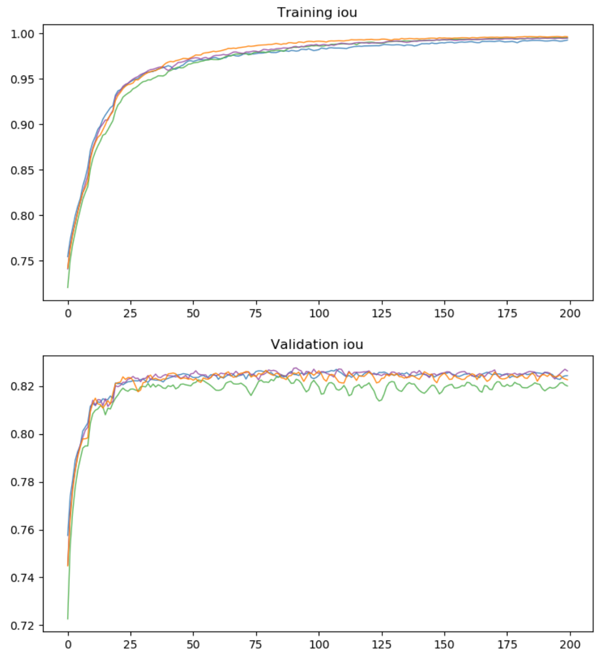


Figure 11: Effects of different losses. See Table 2 for the legend

Loss weights				
l1	edge	const	max val. IoU	color
100	0	0	0.827	blue
100	50	0	0.825	green
100	0	15	0.828	purple
100	15	15	0.827	orange

Table 2: Results of experiment 2

Looking at the graph of the validation IoU, we can see that adding the L_{const} loss, with or without the L_{edge} loss with a small weight, is very similar to just using the L_{l1} loss. Adding only the L_{edge} loss with a larger factor, however, reduces the overall accuracy on the validation set.

7 Conclusion

As a conclusion, we can see that the Unet performs better overall than the GAN and that adding image augmentation does not improve the accuracy. As for the losses, the L_{const} loss seems to not have much effect on the training, whereas the L_{edge} loss seems to be disadvantageous and results in lower validation IoU scores.

Some examples of validation images can be seen in Figure 12 and some more are added in Appendix C. Visually, the results are good but still rough for some characters, and sometimes even really bad for others. This final output is thus not usable as a real character set, but the project does show that it is possible to learn some transformations from rough to refined characters. For the other characters, either there is not enough training data for the network to learn all needed transformation, or it is possible that having only a rough image of the character as input is not enough information and that the model needs to be improved in some way.

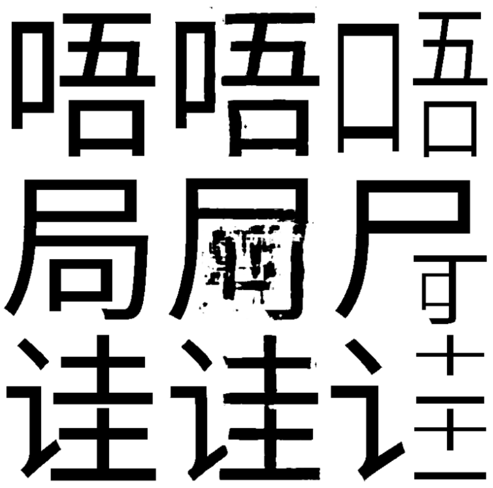


Figure 12: Results on 3 validation characters. Left: target image, middle: output image, right: input image

8 Future work

In this project, the second part was only tested on a single font, but ultimately the goal would be to train on multiple fonts at a time to see if the network is able to generalize enough, so that it could be used for completely new fonts.

The current database holding all the character definitions also contains only around 3000 characters, which should be expanded to the minimal character set of around 6000 characters to be complete. As seen in the first part of the project, precise definitions seem to be superior to general layouts and thus another step would be to define a precise definition for every character and maybe implement a new technique for combining the components together that generalizes even better to different fonts.

Lastly, another idea was discussed during the project, but not implemented. This idea is to train a network on a single character or a group of similar characters at a time, using a lot of different fonts. This could allow the network to really generalize over the different fonts, while not having to learn a lot of different transformations, and this could lead to better results. This idea does involve training a large number of networks and has thus not been done, but is a possible alternative to the solution proposed in this project.

References

[1] Rainer Erich Scheichelbauer. Reusing shapes: Smart components. `https://glyphsapp.com/tutorials/smart-components`, 2015.

[2] Matthew Skala. Tsukurimashou: A japanese-language font meta-family. *TUGboat*, 34(3), 2013.

[3] Yucen Tian. zi2zi: Master chinese calligraphy with conditional adversarial networks. `https://github.com/kaonashi-tyc/zi2zi`, 2017.

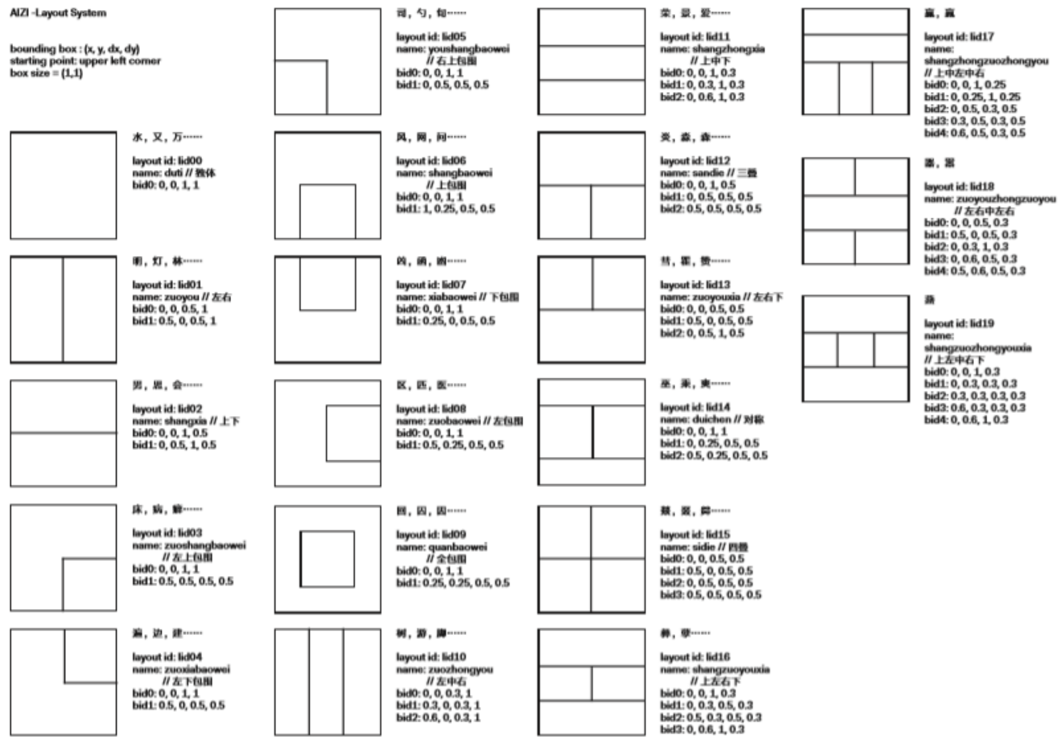
[4] Neill D.F. Campbell and Jan Kautz. Learning a manifold of fonts. *ACM Transactions on Graphics (TOG)*, 33, 2014.

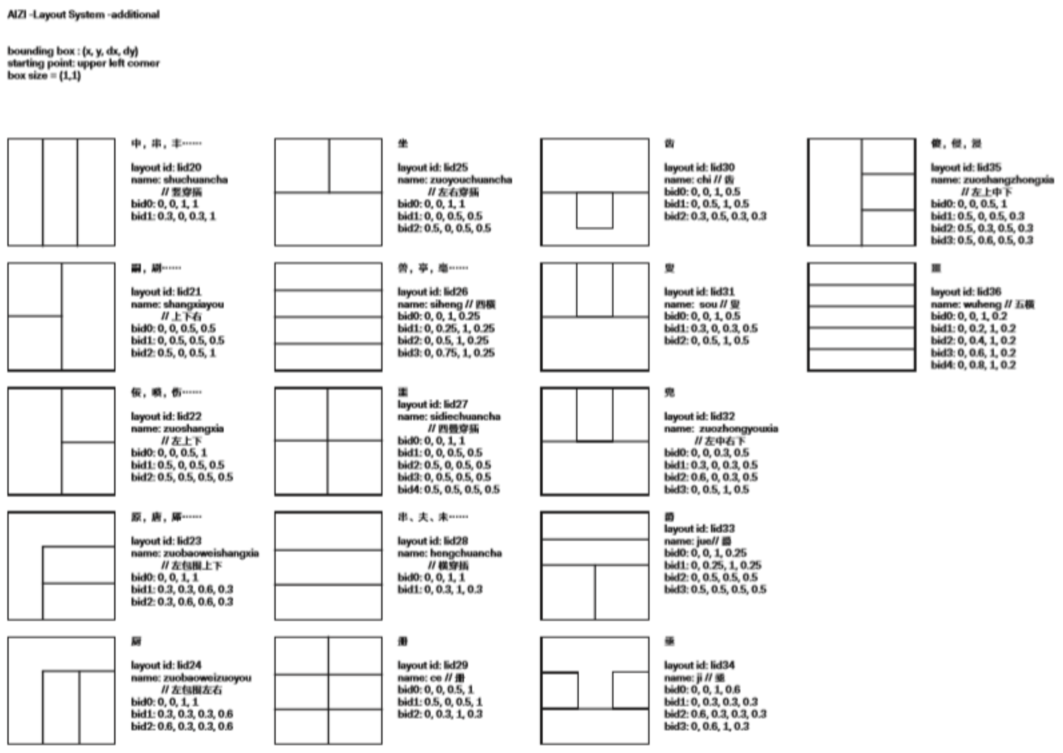
[5] Roland S. Zimmermanna and Julien N. Siemsa. Faster training of mask r-cnn by focusing on instance boundaries. *arXiv preprint arXiv:1809.07069v3*, 2019.

[6] Irwin Sobel. An isotropic 3x3 image gradient operator. *Presentation at Stanford A.I. Project 1968*, 02 2014.

Appendices

A Layouts





B Model architectures

Table 3: Unet architecture

Layer	Type	Parameters
1	convolution	64 filters
2	encode layer	128 filters
3	encode layer	256 filters
4-8	encode layer	512 filters
9-11	decode layer	512 filters, dropout (rate = 0.5)
12	decode layer	512 filters
13	encode layer	256 filters
14	encode layer	128 filters
15	encode layer	64 filters
16	encode layer	1 filters
17	tanh layer	/

Table 4: Encode layer

Layer	Type	Extra information
1	leakyReLu activation	/
2	convolution	Strides of 2 for the downsampling
3	batch normalization	/

Table 5: Decode layer

Layer	Type	Extra information
1	ReLu activation	/
2	deconvolution	/
3	batch normalization	/
4	dropout	Only if specified
5	concatenation	Concat. with the corresponding encoded layer

Table 6: Discriminator architecture

Layer	Type	Parameters
1	convolution	64 filters
2	leakyRelu activation	/
3	convolution	128 filters
4	batch normalization	/
5	leakyRelu activation	/
6	convolution	256 filters
7	batch normalization	/
8	leakyRelu activation	/
9	convolution	512 filters
10	batch normalization	/
11	leakyRelu activation	/
12	flatten	/
13	fully connected	1 node
14	sigmoid activation	/

C Visual results

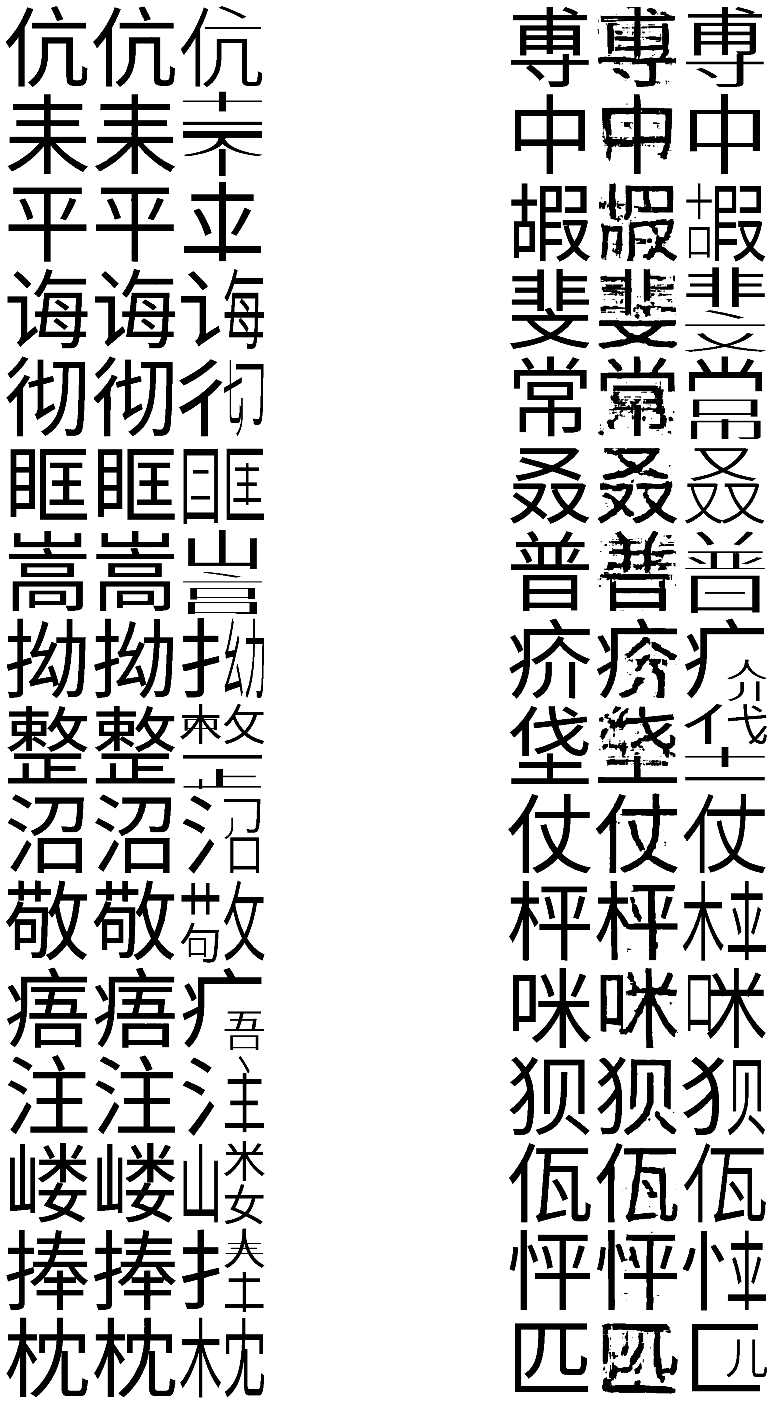


Figure 13: Batches of 16 characters at the end of training. Left: training sample, right: validation sample. For each character, the middle image is the output image, the left image is the target and the right image is the input.