SPLINE TEMPLATED BASED HANDWRITING GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Publicly available labelled handwriting data is fairly limited in its representation of styles as well as in the volume of examples for many topics. We find that trying to use these publically available datasets as training data on unrelated unlabelled handwriting datasets produces unsatisfactory results and would not produce a trained system capable of performing adequately in real world tasks. We propose a method of character and word generation using fonts as templates, as large numbers of handwriting fonts are available for personal use online. Our technique, based on modifying previous work in mechanical handwriting modeling and template based generation and extending that to arbitrary images of letter and words with an automatic method of generating templates through splinification. We find that we get reasonable results on MNIST, EMNIST, IAM, and a proprietary dataset created from Boeing aircraft maintenance forms when no training data is available. This method requires minimal training and generates in a fast, easily parallelizable fashion.

1 INTRODUCTION AND RELATED WORK

Neural networks generally have the highest recognition accuracy, either with purely convolutional neural networks (CNNs) Cireşan et al. (2010) for character recognition, or combining CNNs with recurrent architectures for word level handwriting recognition Puigcerver et al. (2016). While these systems do have very high accuracy, they also demand large in-domain training datasets. Various methods of dealing with this, both in handwriting recognition and text recognition more broadly have been proposed. We break these into two categories: supplementing existing data, and creating wholly new synthetic data.

The simplest method of the first category is simply to create variations of the preexisting training data. Le Cun et. al. LeCun (1998) apply random planar transformations to their training data, while Neuberg Neuberg (2003) takes that a step farther and applies realistic effects you might see on documents such as shadows and creases. Jayasundara et. al. Jayasundara et al. (2019) also goes beyond affine transformations by adding controlled noise to instantiation parameters in capsule networks, then generating from a small amount of in domain images. Prabhu Prabhu et al. (2019) uses a mixture of elastic distortion and GANs applied to synthetic data to distort images.

Beyond distorting images, various methods of using fully synthetic data have been proposed. This is most common in typed text recognition, where if the same font is used a potentially unlimited amount of essentially in-domain data can be generated. Neuberg Neuberg (2003) uses this to generate large amounts of data based off of a curated font set. Jaderberg et. al. Jaderberg et al. (2014) do something similar in creating scene text data. This has been more broadly applied to create handwriting as well. Yamamoto Yamamoto (2003) creates a method of adding realistic looking variation and connection between characters to two Japanese fonts, while Prabhu uses an Indian langauge font set as the basis for their previously mentioned system. While not necessarily used this way, Lake et. al.'s Lake et al. (2015) Bayesian programming based motor program system can also be used to create synthetic data from font templates while Graves's RNN generation system Graves (2013) could be used to create word level data. A more manual approach described by Varga Varga et al. (2005) hand crafts a font set out of splines, then uses the resulting splines as the basis for generation using a Delta LogNormal handwriting model (Guerfali & Plamondon (1995)).

Figure 1: An example of connected fonts

There are two major drawbacks of the previous work. First, many of the ones that focus on fonts use just one or two fonts. Others such as Jayasundara et al's use some numbers of real examples per class (in this case 200). There are much larger repositories of fonts available online, however, well beyond what Varga or Prabhu use. Many of these fonts are labelled as handwriting fonts, which suggests that they were created to mimic human handwriting.

Secondly, they are often constrained in the length of what they can generate-often focusing on just one letter. Lake et. al.'s system includes a random walk over potential stroke orders, which means as the number of segments increases, the amount of time taken in generation quickly becomes unfeasible. Both GANs and Capsule nets have mostly focused on generating letters. Varga's system can be used to create words, but they do that by generating letters, then stitching them together to form words, with heuristics used to generate connected handwriting. Out of all of the systems we evaluated, Grave's RNN based system alone easily generated connected text.

We attempt to mitigate the previous problems by creating a method for fitting splines arbitrarily long sequences of handwriting, then similarly to what Varga did, applying a DeltaLog Normal handwriting model on top of it to generate human-esque variation. What allows this to work over extended sequences is that many of the fonts available can generate connected output, as seen in figure 1. These are then used to generate words directly at the word level. This allows us to generate sequences of arbitrary length, with all of the benefits of Varga's system: minimal training and fast generation to rapidly create synthetic datasets.

We test this model by generating synthetic data and testing on real data across several different handwriting datasets: MNIST LeCun (1998), EMNIST (NIST SD19 digits and letters, formatted in the MNIST format) Cohen et al. (2017), a character level dataset created from Boeing maintenance records, as well as two word level datasets: IAM Marti & Bunke (2002) and similarly a word level dataset created from the same airline records. We run comparisons across datasets to test how well these datasets transfer (say, train on IAM and test on Boeing), as well as comparisons against two other generation methods (BPL and RNNs), with just RNNs for word level comparisons. We also compare against capsule nets on character level data.

We find that our system, when using no in domain data, out performs using out of domain data by up to 30%, and also outperforms the other two generation methods by similar amounts, larger in the case of RNNs. It has comparable results to the capsule nets on digits, even when trained on fully out of domain data, but sees a performance drop of around 10-15% over digits and letters, unless augmented with real examples, where it only sees a slight performance drop. It performs well on top down word recognition, but less well on character level recognition across words.

2 GENERATION

Our generation method follows the following steps:

- Create a large set of handwriting fonts
- Generate the desired word with a given font
- Fit a set of splines to the word
- Convert splines into an ordered set of arcs and lines using segmentation points
- Expand these arcs and lines to generate variation and allow overlap of strokes
- Using the Delta LogNormal method, generate and overlap strokes from the arcs and lines

2.1 CREATING A SET OF HANDWRITING FONTS

First, we curate a large database of handwritten fonts to generate from. We scraped several websites for handwriting fonts, and have found more than 3000 such fonts that were also available for personal use (unfortunately, due to this, we cannot make this dataset available).

This handwriting font database was still quite messy though, so we manually curated it to a smaller database by removing fonts that were obviously not handwriting, that would not produce reasonable skeletons (most prominently hollow fonts, but also fonts that were overly complex). There were also a large number of fonts where uppercase letters were the same as lowercase, and these had to be separated into separate classes so as not to render the upper/lower case distinction meaningless.

This suggests a broader range of styles is available then has previously been utilized unil now, which gives us a much broader range of styles than just one or two, which lets us model a larger range of handwriting.

2.2 CREATING CHARACTER TEMPLATES

After we curate a set of fonts, we then can use them to write words as templates for generation. The first step in this process is to create an automatic method of fitting splines to handwritten character skeletons.

First, we take the letter written in the chosen font and skeletonize it, using the skeletonizer that comes with Lake et al's work, modified by replacing their thinning program with an implementation of the Zhang-Suen Thinning Algorithm (Zhang & Suen, 1984). We use this skeletonization as it does a very good job merging nearby critical points, and find that in practice combining it with the modified thinning algorithm produces the most natural looking skeletonizations.

From here, we fit a series of cubic spline interpolations to each segment. First, a 2d spline cannot always be fitted, as some shapes, such as circles in '0' are not functions. To get around this problem, we fit two splines to both the X and Y values, parametrized with a parameter t, and modeled each spline as a parametric set of two splines x(t) and y(t).

We found that naively attempting to apply spline interpolation resulted in overfitting to the curve-for example, a zero with fifty points in the segmentation would be fit by a thirty part piecewise spline segmentation-not far off from creating a piecewise spline by creating lines between every two points.

To mitigate this problem we start by creating a spline between two points, then recursively add more points and creating a piecewise spline interpolation over that more restricted set of points until a 'goodness of fit' threshold is reached. In this case we stop when 95% of the spline is within the black ink of the original image. To chose the next point we calculate the point that the new set of splines will have the minimum distance between each point in the skeletonization and the set of splines, as illustrated in figure 2.

This greedy approach quickly creates a potential splinification of the given skeleton. In order to refine it, we perform a beam search across potential sets of points, pruning any branches whose number of control points pass the number of points in the original greedy splinification.

We perform this with cubic splines, linear interpolations, and Modified Akima splines Akima (1970), choosing the final result that has the lowest margin. This gives our system some degree of flexibility-for straight characters like 'T', linear interpolations tend to be better, while cubic and Akima splines work better for round characters.

2.3 CREATING AND OVERLAPPING ARCS AND LINES

At this point, our implementation broadly follows a modified version that of Varga et. al. We start by giving the splines an ordering using a similar system to Edwin de Jong's MNIST sequence data (Jong). From the ordered splines, we calculate segmentation points: start and end points, local maxima of curvature, inflection points, and break up curves greater that 180 degrees, while using numeric methods to calculate these at overlap points in the piecewise spline. From this, the splines are converted into a set of arcs and lines, also represented with parametric functions.

Figure 2: A simple example of greedy splinification. Beam search further narrows the band, especially on the left side.

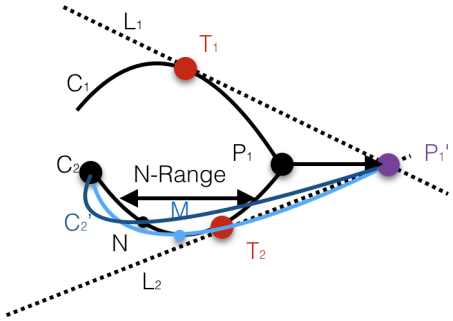

Figure 3: An illustrated example of stroke expansion. T_1 and T_2 are the tangent points selected randomly from the midpoint to target point. P_1' is the intersection point of the new arcs constructed at the intersection of the tangent lines. M is the midpoint, where the window $N-Range$ is constructed from-N is chosen randomly from this range. The dark blue line C_2' is the new curve, while the light blue line in the background is the hypothetical line constructed from the midpoint. Notice this line always widens the stroke (it can easily be shown this curve never intersects the original curve except at the end points and tangent point), while the randomly chosen lines will be narrower at some point.

The next step is stroke expansion. Varga et. al. found that simply generating from this set of splines did not create enough variation, and our experiments broadly agreed with him. At this point, we therefore then expand the strokes in a semi-randomized fashion, to be contracted again during the generation process in the next section.

In our case, for stroke expansion say we have an intersection point of two arcs, a_1 and a_2 and point p. We choose points from a_1 and a_2 randomly between the midpoint of the respective arcs and p. From there, we calculate tangents at those points and take the intersection of the tangent points to be the new stroke expansion point. To form the new, expanded arc, we have the starting point and the new expanded point, but we need a third point to define it. We experimented with taking a random point from the initial arc and by taking the point which makes the expanded arc tangent to the old unexpanded one, but found that the first results in illegible handwriting because of too much randomness and the second handwriting that is not useful for training classifiers because of too little randomness. To address that, we take a random point in a window of 1/4th the total stroke distance around the tangent point. This gives us the expanded stroke, as seen in figure 3.

2.4 DeltaLog Normal Generation

The next step is to generate. In order to do this, we use the Delta LogNormal model of handwriting generation from Guerfali et al (Guerfali & Plamondon, 1995). A Delta LogNormal handwriting generation model models handwriting as two opposing actions: a neuromuscular agonist and antagonist associated with each movement. Each action can be characterized by a LogNormal function, then the difference between them is taken to represent their opposing nature (hence, Delta LogNormal).

We can then calculate the velocity profiles for $v(t)$ as in Guerfali et al:

$$v(t) = D_1 * \Lambda_1(t, t_0, \mu_1, \sigma_1^2) - D_2 * \Lambda_2(t, t_0, \mu_2, \sigma_2^2) \tag{1}$$

where D_i is a lognormal:

$$\Lambda_i(t, t_0, \mu_i, \sigma_i) = \frac{1}{\sigma_i\sqrt{2\pi}(t - t_0)} e^{-\frac{(ln(t-t_0)-\mu_i)^2}{2\sigma_i^2}} \tag{2}$$

where $D_1 - D_2$ is the amplitude of the movement, and μ_i and σ_i are the logtime delay and logresponse time. We then randomly choose the parameters, guided by Guerfali et al and our own experiments to balance useful variation with legibility. In our case, we have the amplitude of the movement, so we choose the ratio of $D1$ to $D2$ to be in the range from 3 to 15, while mu_1 is in the range from 0 to 0.3, with mu_2 being from 0.4 to 0.6 larger than mu_1. σ_i is fixed at 0.5.

We also have to calculate the direction, which as we are following an arc (as we broke the character into arcs and line segments-the case for line segments is trivial) is the integral of the velocity at a given time times the curvature k, plus the initial direction. The change in time is thus:

$$\theta = \theta_0 + k(D1 \int \Lambda_1 - D2 \int \Lambda_2) \tag{3}$$

As Λ_i is a log normal function, we can calculate its integral by evaluating its cumulative distribution function:

$$\int \Lambda_i = \frac{1}{2} + \frac{1}{2}\text{erf}[\frac{ln(t - t_0) - \mu_i}{\sqrt{2}\sigma_i}] \tag{4}$$

where erf is the error function.

This gives us basic velocity and direction profiles for the character, but again we find that that alone does not provide enough diversity in the written text, often creating letters too similar to the original input. In terms of overlap, we shift each stroke by a random factor of up to 1/7th of the total stroke length, and sum the initial directions and velocities at times where multiple strokes overlap. Once velocity profiles are generated, strokes are traced to create thinned character forms. We experimented with various methods of thickening them, including the in filling used in Lake's one shot learning paper, but found that thinning the test words instead gave better results.

Finally, going beyond what been described in most other systems, this model can be used to directly produce word level data by running on typed words, instead of producing characters and stitching them together if necessary. In our case, we find that this was not necessary as our splinification produces good results on the word level.

3 RESULTS

To test our generation, we train our models on both word and character level data, across several datasets. For character level data, we test MNIST, NIST SD19, and a dataset of numbers manually cut out from maintenance records forms from Boeing. The Boeing dataset consists of around 1,200 digits. For word level results we similarly test on the offline IAM handwriting dataset and a set of 500 words manually cut out of the Boeing maintenance forms. Our character level model is the standard LeNet system LeCun et al. (1998) for MNIST and AlexNet Krizhevsky et al. (2012) for NIST, trained under out-of-the-box Digits settings, and on word level we test both word level classification using AlexNet (on a subset of the 500 most common words) as well as character level classification using a CRNN recognition system on the full IAM dataset. An example of characters generated by our system can be found in figure 5, with further examples in the appendix. The synthetic handwriting was generated in parallel using MATLAB on either 2.60 or 2.20 GhZ CPUs with 32 GB of memory, then the recognition systems were trained on a GeForce GtX 1080.

We run comparisons against Graves's RNN handwriting generation system (Graves, 2013), Lake's one shot learning system (Lake et al., 2015), our spline based system, Jayasundra's capsule network, as well as against non-artificial data (that is, systems trained on the MNIST, NIST, or IAM training sets-we do not have enough Boeing examples to train on). For the RNN system, we tested several implementations, and settled on SJ Vasquez's implementation of Grave's system found at

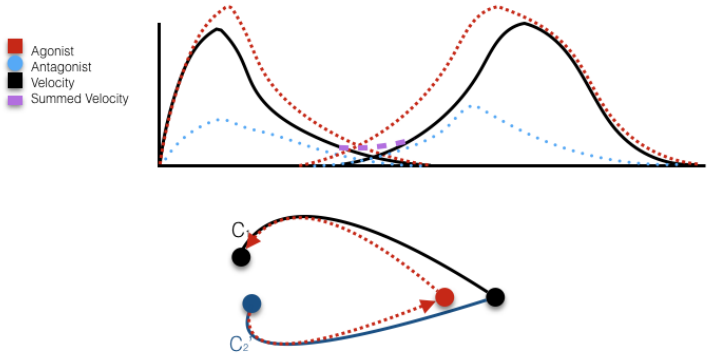

Figure 4: An example of velocity profiles of the two strokes from the previous example, and their overlap. In the upper half, the dashed and circled lines represent the agonist and antagonist strokes respectively, while the black line is the resulting velocity profile. The overlap is calculated simply by adding the two black lines, this is illustrated in the purple dashed line in between the two lines. In the lower half, the result of this overlap is shown in the dashed red arrow-it contracts the previously expanded stroke.

https://github.com/sjvasquez/handwriting-synthesis, as we find this gives the best looking data. We then generate data by randomly sampling given text from the RNN. There is a mismatch in some of the underlying data, as the capsule network was trained on 200 real examples per class and the RNN system was trained on IAM (so we can't compare against it in the case of IAM offline, as it has seen the training data), but at least for Lake's system, we train models off of the same font set, and then generate from there.

We find that on MNIST, as seen in table 3, our system performs the better than the competing artificial generation systems, while still slightly under performing capsule networks and a system trained on the MNIST training data itself. The RNN under performs, which we speculate could be due to the instability of its generation process or to the more limited number of styles in the training set (IAM has fewer than 400 writers). Most of its generated data looks very human, with some examples better than the spline system we present, but it produces less variation and more garbled results than other systems, limiting its utility. One advantage of the spline based system is that by tweaking random variables and varying the geometry, how random the output will be can be adjusted. we find reducing the bias of the neural network gives better results, but we find that despite excellent performance, there is still too much noise and not enough variance.

We see similar results on the Boeing data, also in table 3, but with training MNIST now being outperformed by the spline system, producing similar results to the one shot learning system. This points to the benefits of having a large number of styles, almost ten times more than is present in MNIST (over 1000 vs around 250 unique authors in MNIST).

While we still see decent results on the EMNIST dataset, our accuracy drops off quite a bit compared with MNIST-down to around 82% across letters, and 73% across letters and numbers. This is quite a bit lower than the capsule network, but in this case 200 examples per class adds up to thousands of examples total, for a not insignificant in domain dataset.

On word level data, we compared against the IAM offline dataset as well as a Boeing word level dataset. Here as well we find our system outperforms the RNN system as well as IAM on the Boeing dataset, as well as getting reasonable accuracy on IAM. Unfortunately, our character level accuracy on IAM and Boeing was quite a bit lower than word level top down recognition and a lot lower than when training on in domain data, so that while the overall form of the words is being accurately represented, there still remains work to be done on the details of the word level generation. This suggests that by extending generation we are able to some extent generalize to words as well as characters, which is an improvement over most of the existing systems.

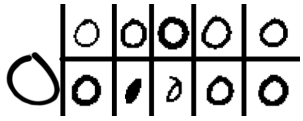

Figure 5: An example of generated zeros. The left is the base font, while the right is ten generated zeros.

	MNIST	Boeing
MNIST	>99%	52%
RNN	67%	32%
BPL	80%	53%
Spline	98%	82%

Table 1: The results on MNIST and Boeing. The left column is what the CNN was trained on, and the top row what it was tested on.

	EMNIST-Letters	EMNIST-Full
Spline	82%	73%
CapsuleNetwork	92%	87%

Table 2: The results on balanced EMNIST letters and full.

	IAM (word)	IAM (char)	Boeing (word)	Boeing (char)
Spline	78%	56%	64%	48%
IAM	81%	94%	$\sim$	33%
RNN	$\sim$	$\sim$	28%	32%

Table 3: The results on IAM. The left column is the results of word level classification accuracy, while the right character level accuracy (which we calculated as 100-character error rate, but presented that way to keep the numbers consistent). We don't run the RNN against IAM, as it is trained on online IAM, and we can't do word level recognition training on IAM and testing on Boeing do to a mismatch in vocabulary.

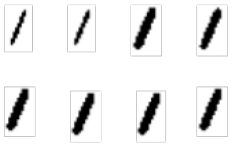

Figure 6: An example of generated ones, the most common case of straight lines-the whole process of adding realistic noise did essentially nothing, and all eight examples look the same.

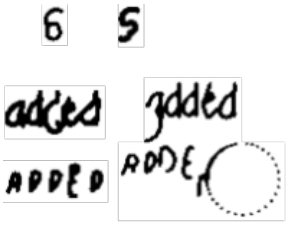

Figure 7: An example of failures involving the generation process. All of these pairs were generated by the same font, and the second example is the failure case. The second six failed to close the loop to the point where it looks like a five, while the second ”added” in the second and third examples has a loop that was flipped in directions, making the character look completely ridiculous. The full loop is cut off in the first example.

4 Error Analysis

4.1 Failure cases and analysis

The system still breaks down in certain cases, and because the generation is based off of splines and geometry, we can pin point the causes of many of these failures.

One set of failure points happens because the generation process breaks, and often takes two forms. The most common of these occurs at the critical point of the loop, as this point is not guaranteed to be closed in the final generation process. While this often happens in human handwriting as well, when it happens in unhuman like ways it can break the difference between two letters, as seen in the set of sixes that look more like fives. The less common but more severe problem is every once in awhile, the third point that is added when forming the arc gets placed on the wrong side of the arc, which results in instead of drawing an arc that is less than 180 degrees, drawing the opposite arc greater than 180 that completes the circle. This can be seen in the case of the lowercase and uppercase ”added” examples, where the ”a” and last ”D” broke respectively and give very unhuman results. The first case could be solved by adding constraints to how ”broken” a loop could become, while the second is trickier-this was originally a large, commonly occurring problem, but a few simple filters removed the vast majority of its occurrences. It is currently at a point where we are satisfied with how rare it is, but further restrictions could make it rarer or remove it completely. These can both be seen in figure 7.

The second failure case happens because despite the greater control given by this spline based method over neural based methods, balancing randomness is still not perfectly done. At times there is too much randomness, especially in word level data, which occurs when small random changes accumulate in the word to the point of being unnatural-any specific character is fine, but the word as a whole looks distorted and unnatural. The opposite often happens when straight lines are fitted to a character, as only the length of the line changes in the stroke expansion/lognormal overlap process as seen in figure 6, as well as in small segments, which naturally can’t change much, which leads to very similar looking characters. Improving the spline fitting process to only contain longer lines and the expansion/contraction process to shift the angle of lines or even add slight curvature may help improve diversity, while tying together random shifts across characters to form a more consistent word could make words more consistent without overly impacting diversity.

5 DISCUSSION

In this paper, we find that applying automatic splinification to font images, then running a handwriting model over them quickly produces relatively good looking artificial handwriting data. By compiling a large handwritten font collection, then testing by training handwriting systems on artificial data then testing on real data, we find that it produces better results than the competing generation systems, and has the added benefit of requiring minimal training, being easy to run in parallel on CPU.

However, the dropoff in accuracy with no in domain data versus using in domain data with this system or capsule networks as well as on character level recognition across words suggests room for improvement. There are a few potential avenues for this. One is by more carefully curating the handwritten font dataset, we may be able to improve accuracy. That would entail both finding more fonts, as well as more carefully removing fonts that don't appear to be handwriting fonts, as well as more carefully curating connected fonts. Another potential approach would be to improve the model. It is likely that finding a way to integrate a neural model instead of the LogNormal model for generation would create better results, if outliers could be pruned and the ability to generate word level data maintained.

It could potentially also be extended to other languages, although depending on language the necessity of collecting large font sets would potentially be limiting in that regard.

ACKNOWLEDGMENTS

We would like to thank Boeing for providing the airline maintenance record database.

REFERENCES

Hiroshi Akima. A new method of interpolation and smooth curve fitting based on local procedures. *Journal of the ACM (JACM)*, 17(4):589–602, 1970.

Dan Claudiu Cireşan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. Deep, big, simple neural nets for handwritten digit recognition. *Neural computation*, 22(12):3207–3220, 2010.

Gregory Cohen, Saeed Afshar, Jonathan Tapson, and André van Schaik. Emnist: an extension of mnist to handwritten letters. *arXiv preprint arXiv:1702.05373*, 2017.

Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

Wacef Guerfali and Réjean Plamondon. The delta lognormal theory for the generation and modeling of cursive characters. In *Document Analysis and Recognition, 1995., Proceedings of the Third International Conference on*, volume 1, pp. 495–498. IEEE, 1995.

Max Jaderberg, Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Synthetic data and artificial neural networks for natural scene text recognition. *arXiv preprint arXiv:1406.2227*, 2014.

Vinoj Jayasundara, Sandaru Jayasekara, Hirunima Jayasekara, Jathushan Rajasegaran, Suranga Seneviratne, and Ranga Rodrigo. Textcaps: Handwritten character recognition with very small datasets. In *2019 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pp. 254–262. IEEE, 2019.

Jong. Mnist sequence data. https://edwin-de-jong.github.io/blog/mnist-sequence-data/. Accessed: 2018-08-17.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

U-V Marti and Horst Bunke. The iam-database: an english sentence database for offline handwriting recognition. *International Journal on Document Analysis and Recognition*, 5(1):39–46, 2002.

Brad Neuberg. Creating a modern ocr pipeline using computer vision and deep learning. https://blogs.dropbox.com/tech/2017/04/creating-a-modern-ocr-pipeline-using-computer-vision-and-deep-learning/, 2003. Blog Post.

Vinay Uday Prabhu, Sanghyun Han, Dian Ang Yap, Mihail Douhaniaris, Preethi Seshadri, and John Whaley. Fonts-2-handwriting: A seed-augment-train framework for universal digit classification. *arXiv preprint arXiv:1905.08633*, 2019.

Joan Puigcerver, Daniel Martin-Albo, and Mauricio Villegas. Laia: A deep learning toolkit for htr. https://github.com/jpuigcerver/Laia, 2016. GitHub repository.

Tamás Varga, Daniel Kilchhofer, and Horst Bunke. Template-based synthetic handwriting generation for the training of recognition systems. In *Proceedings of the 12th Conference of the International Graphonomics Society*, pp. 206–211, 2005.

Kohji Robert Yamamoto. Development of kana typeface generated using stroke data for print media. http://robertyamamoto.jp/stroke_kana_font.html, 2003. Masters Thesis.

TY Zhang and Ching Y. Suen. A fast parallel algorithm for thinning digital patterns. *Communications of the ACM*, 27(3):236–239, 1984.