The Learnable Typewriter: A Generative Approach to Text Analysis (Supplementary Material)

Ioannis Siglidis[0009−0002−2278−5825], Nicolas Gonthier[0000−0002−9236−5394], Julien Gaubil[0009−0006−2577−0847], Tom Monnier[0009−0008−1937−6506], and Mathieu Aubry[0000−0002−3804−0193]

[1] LIGM, Ecole des Ponts, Univ Gustave Eiffel, CNRS, Marne-la-Vallée, France
Correspondence: ioannis.siglidis@enpc.fr

This supplementary material provides additional qualitative results and details on our method and experiments. First we present random results similar to the ones presented in Figure 3 and 4 of the paper, both in the supervised and the unsupervised setting. Second, we detail the Gaussian pooling of the encoder features and the spatial transformation of the sprites onto the target canvas. Third, we detail the procedure of extracting exemplars and comparing with SSIM on MFGR, and show examples of exemplar diversity. Fourth, we present the algorithm and the methodology we have used to associate sprites to characters for our quantitative evaluation in the unsupervised setting. Finally, we detail the way we adapted MarioNette [7] and DTI-Sprites [5] to construct unsupervised baselines for our task.

1 Additional results

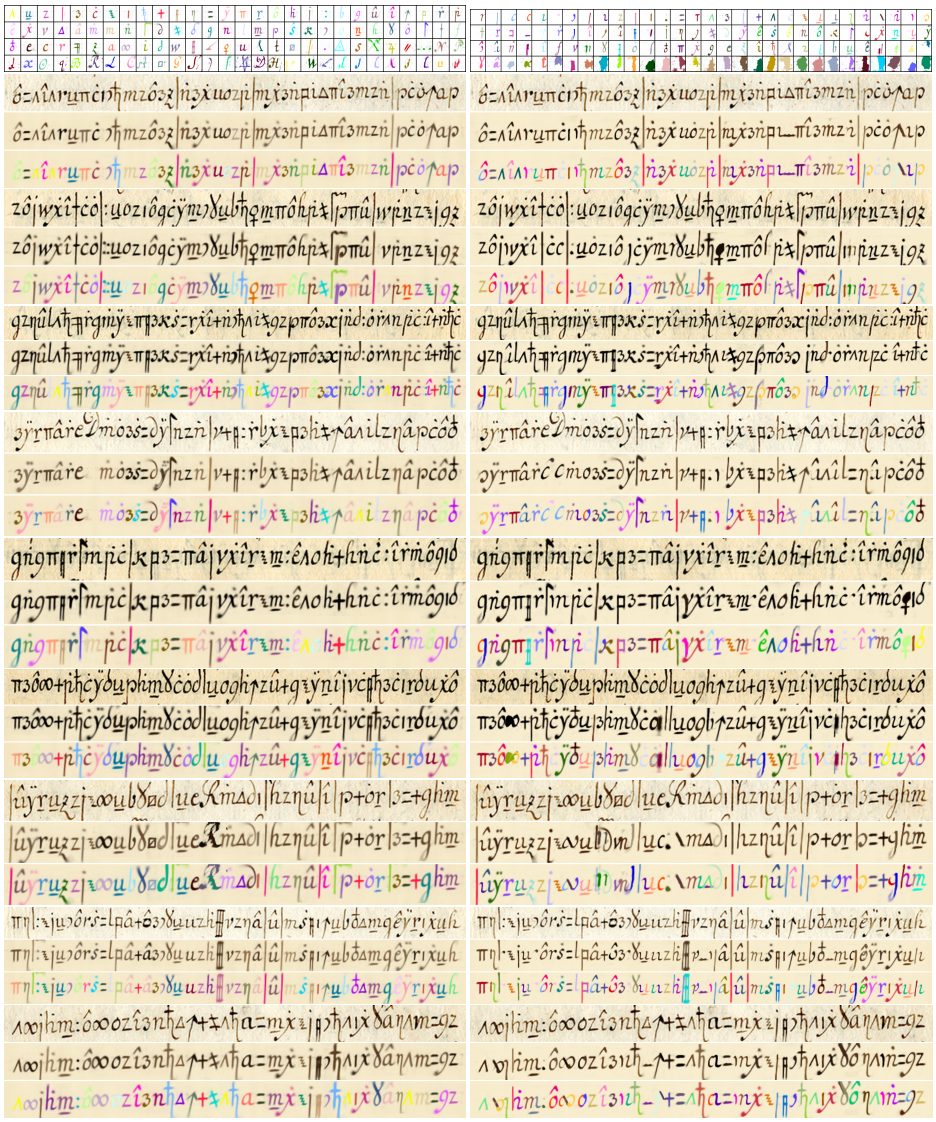
We provide further randomly drawn examples of reconstructions both for Google1000 Figure 1 and for the Copiale cipher Figure 2 from their respective test sets for both the supervised and the unsupervised version of our method. In the beginning of each figure we locate the sprites learned by each method and to which unique colors have been assigned. Below it follow triplets of ground-truth images, reconstructed images and colored segmentations (with the same colors as the ones displayed in the beginning) for a set of randomly drawn examples for each dataset. Note that really similar characters will be assigned to different colors if they are reconstructed by different sprites.

2 Method details

Gaussian Pooling In order to compress (channel-wise) the two dimensional $H/4 \times W/4$ output of the encoder to a one dimensional $W/16$ vector, we use a Gaussian pooling. Concretely, we perform the convolution of the output with a Gaussian kernel κ of size $H/4 \times 4$, with no padding and with a horizontal stride of 4. The kernel κ is defined $\forall (i,j) \in [1, H/4] \times [1,4]$ as $\kappa[i,j] = \hat{\kappa}[i,j] / \sum_{ij} \hat{\kappa}[i,j]$,

(a) Supervised (b) Unsupervised

Fig. 1: **Random results on Google1000 [8] with and without supervision.** The top of the figure shows the sprites learned by our method, colored as in the semantic segmentation. Then, for each input line (top) we show the reconstruction provided by our method (middle) and the corresponding semantic segmentation (bottom). This figure extends Figure 3 of our paper.



(a) Supervised (b) Unsupervised

Fig. 2: Random results on the Copiale cipher [4]. The layout is the same as Figure 1, extending Figure 4 of our paper.

where:

$$\hat{\kappa}[i,j] = \exp\left[-\frac{1}{2}\left((i - {}^{H}/_{8})^2 + (j-2)^2\right)\right] \tag{1}$$

Sprite Positioning We parametrize the scale and translation transformations by a scaling parameter σ and a 2D translation vector $\tau \in \mathbb{R}^2$. We define the affine transformation matrix M from the sprite space to the image space, using image coordinate systems centered at the middle of each feature's receptive field and a sprite coordinate system centered at the middle of the sprite:

$$M = \frac{H}{h}\begin{bmatrix} \frac{1}{\sigma} & 0 & \frac{\tau_1}{2} \\ 0 & \frac{1}{\sigma} & \frac{\tau_2}{2} \end{bmatrix}, \tag{2}$$

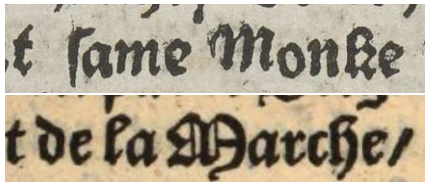
where h is the height of the (square) sprites, H is the height of the line. We obtain the values for isotropic scaling σ and 2D translation τ as the output of the transformation network t_θ (three out of six outputs, the three other being the color of the sprite) to which we apply different non-linearities. For the translation τ, the two corresponding outputs of the linear layer are clamped between -1 and 1. For the scale σ, we apply a non-linearity $x \to \exp(x)$ to the corresponding output of the linear layer.

3 Extracting and Comparing Exemplars on MFGR.

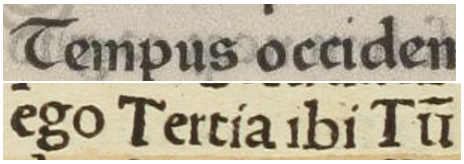
To extract exemplars from MFGR [6], we sort text lines for each character according to its descending amount of appearance and mask & extract the first that looks visually clear. Note, however, that as mentioned in the main paper, for several fonts and certain characters there a more than one exemplars that are needed to visually summarize them. We support this observation with visual examples in Figure 3. To remove each exemplar from its background, we use GHT [1] to binarize it and then multiply it with its binary mask. Then, we normalize the mask and resize & center it with a constant aspect ratio on a canvas of 48 x 48 with a margin of 10 pixels. In order to compute SSIM, we also threshold the ground truth at an intensity of 0.4 and resize & center it in the same way. To improve the comparison, we also remove unwanted noise by extracting the main connected component in both the sprite and the g.t. exemplar using [2,3].

4 Unsupervised Evaluation

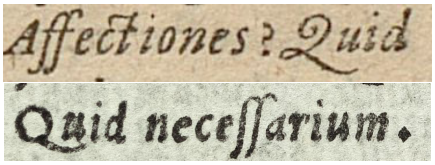
A simple way to quantify the performance of the unsupervised version of our method is to assign sprites to characters and use this association to perform OCR/HTR. Note that in the supervised case this assignment is already known since we associate a sprite to each character at the beginning of training. In practical scenarios, in the unsupervised setting, the association between the sprites and a characters could be performed by a user. Instead, for quantitative evaluation, we want to perform this association automatically, and using only



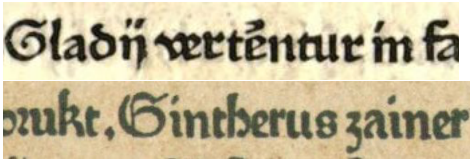
(a) Bastarda (M)



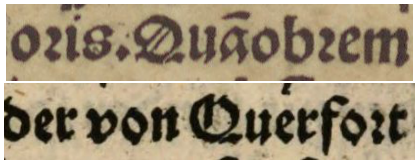
Gotico Antiqua (T)



(b) Italic (Q)



Gotico Antiqua (G)



(c) Schwabacher (Q)

(d) Gotico Antiqua (E)

Fig. 3: **Character Diversity in MFGR [6].** Characters may have both big (a), (b) or slight (c) distinct visual variation. In Gotico Antiqua, this happens to multiple characters with the extreme case of E, which appears in at least 3 visually distinct forms (d).

annotated text lines. Since we have to handle sequences of variable lengths, since our predictions include errors, and since the mapping between sprites and characters is not necessarily bijective, obtaining an optimal assignment is challenging. In this section, we propose a simple algorithm to find an approximate solution.

Formalization. We assume that each sprite corresponds to a character (i.e. we do not consider cases such as a sprite corresponding to only part of a character or a sprite including two characters). Our problem can thus be formulated as the optimization of an assignment matrix $\tilde{A} \in \{0,1\}^{K \times N}$ where K is the number of sprites and N is the size of the alphabet, and where for each sprite s there is a unique character c for which $A_{s,c} = 1$, when sprite s corresponds to character c, and zero otherwise. We relax the problem by optimizing instead a matrix $\tilde{A} \in [0,1]^{K \times N}$ which we parametrize as the column-wise softmax of a matrix A, i.e. $\tilde{A}_{s,\cdot} = \text{softmax}(A_{s,\cdot})$. We will compute the final assignment matrix $\tilde{A}$ by associating to each sprite s the character c for which $A_{s,c}$ is maximal. We

Algorithm 1 Learning Sprite Alphabet Correspondences

Input: Y ▷ **g.t. (characters), pred. (sprites) pairs**
Output: $\tilde{A}$ ▷ **optimal assignment**
Initialization: $A \sim \mathcal{U}(0,1)^{K \times N}$ ▷ **assignment matrix**
for $(y,\ \hat{y}) \in Y$ **do**
  Find the optimal alignment between y **and** $\hat{y}$ **given** A
  Perform a gradient step to minimize $\mathcal{L}_M(y,\ \hat{y},\ A)$
return $\text{argmax}(A,\ axis = 1)$

assume that we are given as input a collection Y, where each element is a pair of a ground truth sequence of characters y and a predicted sequence of sprites $\hat{y}$.

Algorithm overview. We propose an iterative mapping optimization scheme, outlined in Algorithm 1. Given a pair $(y, \hat{y}) \in Y$ and an assignment matrix A we define a matching loss $\mathcal{L}_M(y, \hat{y}, A)$. Our algorithm, iterates over pairs $(y, \hat{y}) \in Y$ and for each pair, it computes the loss as a function of A and then performs a gradient step on A. After the optimization has converged, we extract the mapping by applying an argmax operator to A in the dimension corresponding to the alphabet. As a final refinement step, we ignore sprites that work as visual wild-cards, i.e., sprites that only improve reconstruction without corresponding to any character. To do so, we sort the sprites by increasing usage frequency, iterate over each sprite and discard it if removing it improves the character error rate in the training set.

Matching Loss. Given a matrix A, associated to a matrix $\tilde{A}$, we define a cost matrix C for associating sprites and characters as $C = 1 - \tilde{A}$ and a cost C_{skip} for skipping a character or a sprite when aligning two sequences. The matching loss $\mathcal{L}_M(y, \hat{y}, A)$ is defined as a minimum cost of alignment between y and $\hat{y}$.

We define the cost of a given alignment between a character sequence and a sprite sequence as the sum of the cost of all the sprites to character associations in the alignment added to the sum of the cost C_{skip} for all the characters and sprites skipped in the alignment. Intuitively, as illustrated on Figure 4, one can see an alignment between a sprite and a character sequence as path in a 2D grid, where the two dimensions of the grid are associated to the sequence of sprites $\hat{y}$ and characters y extended by start symbols which we use to initialize the association between the two sequences. Diagonal displacements in the grid are interpreted as an association between sprites and characters, and horizontal or vertical displacements as skipping sprites or characters.

One can thus compute the cost of an alignment by moving along the associated path from the top left corner of the grid to the bottom right which can be decomposed to a sequence of two types of steps: (1) a diagonal step to the bottom right direction toward a cell associated to sprite s and character c, that corresponds to matching the sprite and character and is associated to a cost $C_{s,c}$, and; (2) a horizontal or vertical step, that corresponds to skipping either an

Algorithm 2 $\mathcal{L}_{\mathrm{M}}(y, \hat{y}, A)$ Matching Loss Computation

Input: $y, \hat{y}, A$ ▷ **g.t., pred., assignment matrix**
Constants: $C_{\text{skip}} = 1$ ▷ **skipping cost**
Output: The computed *Matching Loss*
$C = 1 - \text{softmax}(A), D = \infty^{|y| \times |\hat{y}|}$ ▷ **initialization**
for $i, j \in \{0, ..., |y|\} \times \{0, ..., |\hat{y}|\}$ **do**
 if $i \overset{?}{=} 0 \vee j \overset{?}{=} 0$ **then**
 $D[i, j] = (i + j) \cdot \mathrm{C}_{\text{skip}}$
 else

$$D[i,j] = \min \begin{cases} D[i-1, j-1] & + C[y_{i-1}, \hat{y}_{j-1}] \\ D[i-1, j] & + C_{\text{skip}} \\ D[i, j-1] & + C_{\text{skip}} \end{cases}$$

return $D[|y|, |\hat{y}|]$

element of the sequence y or of the sequence $\hat{y}$ and is associated in both cases to a cost C_{skip}. An optimal alignment between the sequences corresponds to a path from the top left position of the grid to the bottom right position (like the one colored green in Figure 4) which has the minimum cost. Such an optimal path can be computed through a dynamic programming algorithm, which we detail in Algorithm 2. We use $C_{skip} = 1$.

Training Details All sprite-character matching models have been trained with a learning rate of 1, a batch size of 256 and for a total of 5 epochs using standard stochastic gradient descent. In practice we don't notice major differences after 1 epoch.

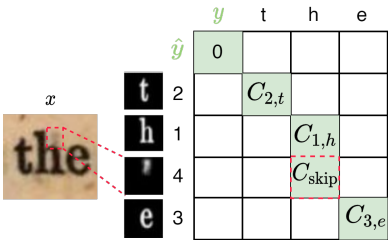


Fig. 4: **Sprite-Character Matching.** An example of a case of misalignment due to a transcription error (in red). In this example the optimal place to "skip" is after sprite 1 as in all the other cases the cost will be higher. With green we denote the best path for which: $\mathcal{L}_M(y, \hat{y}, A_{s,c}) = C_{2,T} + C_{1,h} + C_{\text{skip}} + C_{3,e}$.

(a) Marionette [7] (b) DTI-Sprites [5]

Fig. 5: **Baseline Sprites.** Left: sprites from MarioNette, for the best model: n_objects = 4 and layer_size = 2. Right: Sprites from DTI-Sprites, both trained on Google1000.

5 Baseline

To construct a baseline for our unsupervised method, we examined two available sprite-based methods on Google1000: DTI-Sprites [5] and MarioNette [7]. We train both methods on square crops of 64×64 crops from our line images using the original code provided by the authors with $K = 80$ sprites. We show the resulting sprites in Figure 5. MarioNette didn't produce satisfying sprites, which demonstrates the changes we propose in our paper are critical. On the contrary, DTI-sprite produced good looking sprites and so we applied it repeatedly on line crops and aggregated the results to produce a baseline for our approach (reported in Table 1 in the paper).

MarioNette [7]. MarioNette splits the input image into a grid and predicts sprites and transformations around each position of this grid. Its main hyperparameter is the layer_size parameter defining the resolution of this grid, which also implicitly defines the sprite-size as a rectangle with height $^{2H}/_{\text{layer\_size}}$ (where 2 accounts for the overlap). At each position of the grid we have a maximum of n_objects layered objects. Finally the background can either be constant (inferred as the color of the largest class when applying K-means on the color space) or learned (i.e. a parameter of the overall architecture).

We tune the hyperparameters of MarioNette by trying layer_size $\in \{2, 4, 8\}$, n_objects $\in \{3, 4\}$ and either a learned or a constant background. Our best results, obtained with a constant background, n_objects = 4 and layer_size = 2

can be seen in Figure 5a. While we have letter-like sprites, the model is not able to well-separate background from characters.

DTI-Sprites [5]. DTI-Sprites reconstructs the input canvas by layering a sequence of sprites in front of a background. We freeze the color of sprites and use $L = 4$ layers. As can be seen in Figure 5, the learned sprites are of high quality, even if a lot of the sprites are duplicates of common letters.

There is no trivial mechanism to extend DTI-sprite to variable-size images. Moreover, the number of possible layers is strongly limited by the computational cost of sprite selection which is exponential in the number of layers. To produce results on complete lines we thus simply concatenate the predictions from crops across the whole line. In case sprites intersect with the border of two nearby crops we select the one which has the largest mask inside each crop. We also ignore any sprite with more than 90% of its total mask overlayed by another sprite.

References

1. Barron, J.T.: A generalization of otsu's method and minimum error thresholding. ECCV (2020)
2. Bolelli, F., Allegretti, S., Baraldi, L., Grana, C.: Spaghetti labeling: Directed acyclic graphs for block-based connected components labeling. IEEE Transactions on Image Processing (2019)
3. Itseez: Open source computer vision library. `https://github.com/itseez/opencv` (2015)
4. Knight, K., Megyesi, B., Schaefer, C.: The Copiale Cipher. In: Proceedings of the ACL Workshop on Building and Using Comparable Corpora (2011)
5. Monnier, T., Vincent, E., Ponce, J., Aubry, M.: Unsupervised Layered Image Decomposition into Object Prototypes. ICCV (2021)
6. Seuret, M., van der Loop, J., Weichselbaumer, N., Mayr, M., Molnar, J., Hass, T., Christlein, V.: Combining ocr models for reading early modern books. ICDAR (2023)
7. Smirnov, D., Gharbi, M., Fisher, M., Guizilini, V., Efros, A.A., Solomon, J.: MarioNette: Self-Supervised Sprite Learning. NeurIPS 2021 (2021)
8. Vincent, L.: Google Book Search: Document Understanding on a Massive Scale. ICDAR (2007)