Texture for Colors: Natural Representations of Colors Using Variable Bit-Depth Textures (Machine Learning 2022 Course)

Vsevolod Avilkin [1] **Mikhail Fedorov** [1] **Svetlana Pavlova** [1] **Victor Adamovich** [1] **Artem Vergazov** [1]

Abstract

Numerous methods have been proposed to transform color and gray-scale images to their single bit-per-pixel binary counterparts. When the resulting binarized image is intended for human viewing, aesthetics must also be considered. Binarization techniques, such as half-toning, stippling, and hatching, have been widely used for modeling the original image's intensity profile. In this report We try to replicate the results of the original paper where authors present techniques to ensure that the textures created are not visually distracting, preserve the intensity profile of the images, and are natural in that they map sets of colors that are perceptually similar to patterns that are similar. The approach uses deep-neural networks and is entirely self-supervised.

Github repo: https://github.com/m-fedorov-s/texture-for-colors
Video presentation: https://drive.google.com/file/d/1mK3H4Se8-SrLQNpmThgkU1lwD0l-8tuZ/view?usp=sharing

1. Introduction

For more than half of a century, numerous methods have been pro-posed to transform images to their binarized counterparts — where each pixel is represented by a single on/off bit [1]. At the highest level, these approaches can be categorized into two sets: those intended for human viewing and those intended as a pre-processing step for one specific task. When created for human-viewing, the images created are typically more aesthetically pleasing and maintain a broader set of the original image's attributes at the expense of losing specific details useful for specialized analysis. Common techniques include stippling, hatching and edge-based approaches. Early examples of cross hatching in Western art, where hatches and cross hatches were used to represent a scene's most salient image colors and features can be traced back to the Middle Ages [2].

Beyond artistic interest, binarization is commonly used in a variety of scenarios. Popular electronic-book readers [3], such as Kindle [4], Boox [5], and Nook [6], as well as public digital signage [7,8] all use e-ink with varying levels of bit-depth. Binary texture images are the basis for a large set of subtractive fabrication processes. These include water-jet, laser-cutting with thin materials(paper, cloth, etc.) [9], or popular home cutting devices [10].

Two fundamental challenges must be addressed to create an effective system. Beyond the difficulties associated with the number of unique textures required to represent each possibility, we cannot utilize a unique texture for every pixel, as each is given only 1-bit in the binarized image. Instead, our method selects which colors to represent based on the prevalence of colors in the images we wish to model (e.g. natural images). It also learns to adaptively combine spatially close pixels in the original image to create a texture representative of localized regions.

Our contributions are as follows:

1. We replicate the results of the original paper on STL10 dataset.
2. We use gumbel-softmax activation function in addition to discretized $tanh$ activation functions

2. Related work

The closest work to ours in terms of motivation is Color2Gray. Color2Gray attempted to handle the isoluminant variations that were not preserved with standard color to gray-scale conversions through explicitly analyzing chrominance and luminance differences. Like [25], we handle isoluminance; however, we employ an information-preservation perspective and dramatically reduce bitrates. Similar problems have arisen in numerous specialized ap-

[1]Skolkovo Institute of Science and Technology, Moscow, Russia. Correspondence to: Alexander Korotin <a.korotin@skoltech.ru>.

Final Projects of the Machine Learning 2020 Course, Skoltech, Moscow, Russian Federation, 2020.

plications that require converting photographs to stylized bi-tonal renderings. In terms of the approach taken, the closest work is.They use a CNN to convert color to gray-scale and back; the work parallels the deep-steganography mentioned earlier. Their architecture is a sub-component of the one presented here. For example, to achieve 1-bpp while maintaining smooth transitions in colors, their method requires augmentation with both the novel loss functions and the architectural mechanisms that will be presented.

A note about the relationship of this work to style transfer, GANs, and Image-to-Image Translation. Our initial approaches to synthesizing a binary image from a full color image was to use GANs within the unsupervised image-to-image translation process and numerous style-transfer approaches.Though good results were sometimes obtained, there were severe drawbacks. With style-transfer, the same color on different original images were not consistently mapped to the same textures – an intuitive expectation. Second, in order to ensure that the result was 1-bpp, the majority of the contributions presented here, the architectures and loss functions, were needed to supplement all the approaches. Finally, the GAN training process was significantly less stable than using explicit information-preservation as the basis of our objective function. The final system described here did not require an adversarial teacher. This yielded a simpler and more stable to train system. It also provided consistency in color mappings across images, and generated at least equal, and most often superior, results.

3. Algorithms and models

Link to GitHub code

First let us describe the general pipeline of the algorithm.

For prepossessing, we apply linear normalization to the data to convert the original images to the tensors with zero mean and unitary standard deviation. It is an important step that helps stabilize the learning process. It is omitted in the original paper, so we decided to do it as an experiment.

The algorithm presented in the original paper consists of an encoder and a decoder. We use PyTorch modules to implement them.

The encoder part of the work is to reduce the dimensionality of an input image by passing the input through a series of non-linear transformations, including a low-dimensional "bottleneck" layer. The decoder part of the algorithm recreates the original image from the internal compressed representation emitted by the bottleneck.

Through the back-propagation of errors between the original input and its reconstruction after the decoding, the encoder learns to compress information about the original image through the bottleneck. Simultaneously, the decoder learns to use that representation to reconstruct the image.

We reduce the representational capacity by generating binary images. This means that each pixel can only represent two values: $\{0,1\}$.

Now let us get at a lower implementation-level description of the algorithm. We used a neural network consisting of three PyTorch modules: PreEncoder and DownDiscretizationEncoder for the encoding part, and Decoder. All three of them are in essence just sets of convolution layers with different parameters and non-linear activation functions.

The original image is a full 24-bpp image. It is transformed by the Pre-Encoder to an internal representation of $128 \times n \times n$ floating point numbers.

The reduction of colors to binary is accomplished within the Down-Discretization Encoder module. It takes the internal representation of the image from the pre-encoder and carries it through a series of convolution layers with a non-linear activation (discretized-tanh), each of which reduces the number of bits per channel in the image until it becomes a 1bpp image. It does so in such a way that different colors of the original image are mapped to different textures in the result image. The emergence of different textures is guaranteed. Otherwise, the encoder would not fit the reconstruction loss specifically designed for that problem. By using convolution layers that allow nearby pixels to be considered, local color information of different regions is transmitted to texture information.

The decoder reconstructs the image from the binary representation. It is used in evaluating the error between the reconstruction and the original during the training procedure.

An important feature of this algorithm is that it is completely self-supervised. The training dataset does not require labeling and there is no need to split data into train, validation and test parts because the loss function is calculated and optimized within each step for each image separately. This gives us some freedom in terms of choosing the dataset - we will develop on this topic in the following sections.

Table 1. Network architecture.
Each layer uses (5×5) convolutions.

Stage	input	layers, channels per layer	activation function	activation function in last year	ouput
Pre Encoder	24 bit n × n	10, 128	relu	relu	128 float n × n
Down Discretization Encoder	128 float n × n	8, 1	tanh discrete$_{256}$-discrete$_4$	tanh discrete$_2$	1 bit n × n
Decoder	1 bit n × n	2, 128	relu	tanh	24 bit n × n

3.1. Loss function

The loss function is a very important part of the autoencoder which initially looked like $L_2(I_{RGB} - I'_{RGB})$, where L_2 is an L_2 norm (we used mse_loss from torch for this purpose) and I_{RGB} is a tensor representing 24bit $n \times n$ image.

$$Error = L_2(I_{RGB} - I'_{RGB}), \tag{1}$$

3.1.1. Relative Intensity Constraints

Sometimes during the encoding the black could represent the bright colors of the original image and white represent dark colors. To solve this issue an addition to loss function is introduced:

$$L_{relative\_intensity} = |tanh(A - A^T) - tanh(B - B^T)|, \tag{2}$$

where matrix A is the brightness of original image and B is the brightness of encoded (1-bit) image. $A - A^T$ helps to deduce the relative brightness of image regions.

3.1.2. Color Continuity Constraints

The second encountered problem is inconsistency of a texture corresponding to colors similar to each other. To solve this problem The difference between encoded images of the original image and of slightly distorted image is calculated as a part of loss function:

$$L_{color\_continuity} = |encode(I) - encode(M(I))| \tag{3}$$

where $encode_2(I)$ is an encoded image of the original I and $encode(M(I))$ is an encoded image of perturbed image $M(I)$. Image was perturbed with $transforms.ColorJitter(brightness = 0.1, contrast = 0.1, saturation = 0.1, hue = 0.1)$

3.1.3. Total loss function

So the total loss function consists of the three above with chosen coefficients α, β
Total$_{Error} = L_2(I_{RGB} - I'_{RGB}) + \alpha \cdot L_{relative_intensity} + \beta \cdot L_{color_continuity}, \alpha = \beta = 0.1$

3.1.4. Implementation of gumble softmax

Gumbel Max trick is a technique for discretization with taking argmax

$$G_i = -log(-log(Uniform(0, 1)) \tag{4}$$

Using Softmax instead of argmax with temperature with Gumbel. Tau(τ) is the softmax temperature parameter which allows us to control how closely the Gumbel-softmax distribution approximates,in experiment $\tau = 1$.

$$y_i = \frac{\exp\left(\frac{\log(\pi_i)+g_i}{\tau}\right)}{\sum_j \exp\left(\frac{\log(\pi_j)+g_j}{\tau}\right)} \tag{5}$$

4. Experiments and results

Figure 1. Results obtained for 2 epochs and loss function with only Relative Intensity Constraints
The first row corresponds to the initial images.
The second row corresponds to binary images with texture.
The third row is a batch of reconstructed images.
You can clearly see the main details of the image in the binarized state.

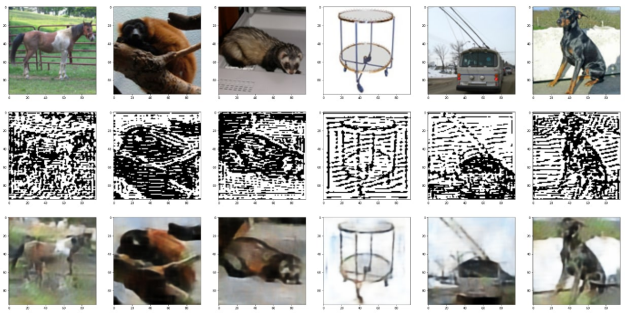

Figure 2. Results obtained for 4 epochs and loss function with both Relative Intensity Constraints and Color Continuity Constraints
The first row corresponds to the initial images.
The second row corresponds to binary images with texture.
The third row is a batch of reconstructed images.
You can see the main details of the image in the binarized state.

For our experiment, we implemented the neural network in PyTorch according to the architecture described above. We trained our model on the STL-10 dataset which contains 100000 unlabeled images. We chose this dataset instead of well-known CIFAR-10 in order to make use of the fact that the algorithm is unsupervised and no labels are required for training. Images for unsupervised learning datasets come from a broader spectrum of sources and have no constraint of

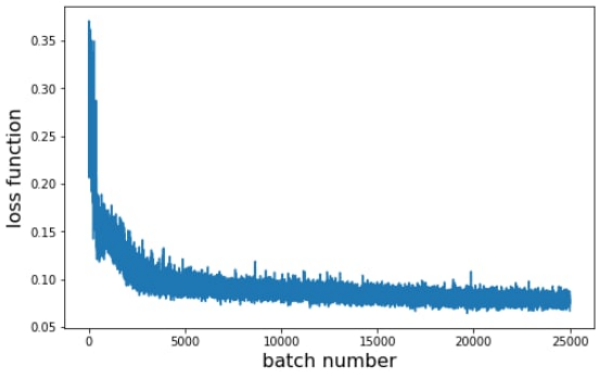

Figure 3. Graph of Loss function with both constraints for 4 epochs

```
[1] loss: 0.008635425008833408
[2] loss: 0.008639723062515259
[3] loss: 0.008626350201666355
[4] loss: 0.008635981008410454
Finished Training
```

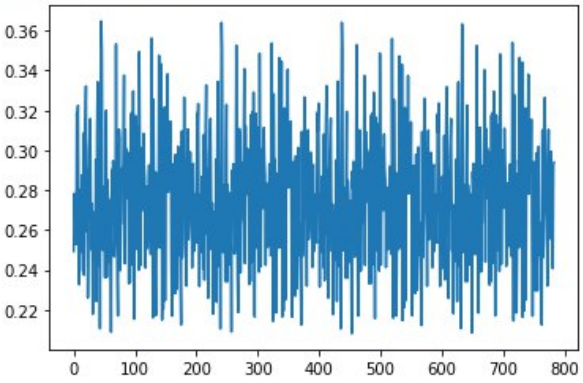

Figure 4. Graph of loss function without gumble softmax for a truncated dataset of 197 images from STL-10

belonging to a specific class. For example, they can contain other types of animals or vehicles, not just 10 classes that CIFAR is limited to.
The training of the auto-encoder was done in Kaggle using GPU as accelerator.

The results of the training are presented in the Figure 1 with the necessary comments. It can be clearly seen (Figures 1,2) that the encoder has been taught to transfer the color contrasts with different textures and the decoder has learned to transform them back to the original with reasonable quality.

It can be said that we managed to replicate the results of the original paper to some extent and obtained similar observations: the good-looking quality of the 1bpp images and the similarity between the decoded images and the original

```
[1] loss: 0.009012298658490181
[2] loss: 0.008501882664859295
[3] loss: 0.006136199925094843
[4] loss: 0.00442919647321105
Finished Training
```

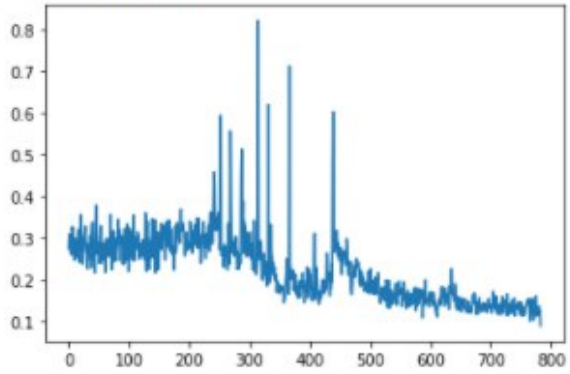

Figure 5. Graph of loss function with gumble softmax for a truncated dataset of 197 images from STL-10

ones. Although the loss function decreases very slowly with epochs (Figure 3), so for a visible improvement in quality of the 1bbp images a lot of time is needed (for each epoch 1 hour of training is needed). After comparing results of the original article and our project it can be noticed that 1bpp image with texture looks better on an images with bigger dimensions.

4.0.1. Implementation of gumble softmax

As an experiment for improving the results, we used Gumble-Softmax trick and trained the model on smaller STL10 dataset - we took only 197 input images to confirm the reasonableness of using discretization techniques. Gumble-Softmax function was performed in forward method after encoding steps. The Figure 4 illustrates the results of training without discretization and we can notice stable value of loss function during 4 epoch equal approximately 0.009. The figure 5 illustrates the results of training with discretization, and the results of loss function decreases from 0.009 to 0.004 during 4 epoch. So, the aim of using Gumble-Softmax is meet.

5. Conclusion

As a result, we reproduced the results of the work on an self-taught method to replace a color or grayscale image with a set of binary textures that represent the original image's colors and patterns. We implemented a deep neural network, which replaces an image with its binary texture pattern and reproduces the original image from it.

We also seelcted a suitable unlabeled dataset which would provide us with such variety of high quality images that labeled datasets cannot provide.

Contrary to the authors of the original paper, we introduced prior transformation to the images - we normalized them. In the paper, the authors mentioned they did no transformations to the images and left it an open topic for further research. In this regard, we managed to obtain reasonable results having added one layer of complexity to the model.

6. References

Baluja, Shumeet. (2021). Texture for Colors: Natural Representations of Colors Using Variable Bit-Depth Textures. arXiv:2105.01768

[1]L. S. G. Kovasznay and H. M. Joseph. 1955. Image Processing. Proceedings of theIRE 43, 5 (1955), 560–570.

[2]Wikipedia. 2020. Hatching. https://en.wikipedia.org/wiki/Hatching Wikipedia:The Free Encyclopedia. Accessed 20/3/2021.

[3]GoodEreader. 2020. GoodEReader. https://goodereader.com/blog/category/electronic-readers. Accessed 20/3/2021.

[4]Amazon. 2020. Kindle. https://www.amazon.com/Amazon-Kindle-Ereader-Family/b/?ie=UTF8node=6669702011. Accessed 20/3/2021.

[5] Onyx. 2020. Boox. https://www.boox.com/en/. Accessed 20/3/2021.

[6]Barnes and Noble. 2020. Nook. https://www.barnesandnoble.com/b/nook/$_{/}N$ $-$ $1pbl.Accessed20/3/2021$.

[7]Wikipedia. 2020. E Ink. https://en.wikipedia.org/wiki/E$_{I}nk.Wikipedia$: $TheFreeEncyclopedia.Accessed20/3/2021$.

[8]eInk.com. 2020. Indoor Large Area Signage. https://www.eink.com/signage.html?type=applicationid=9. Accessed 20/3/2021.

[9] Glowforge. 2020. Glowforge. https://glowforge.com/. Accessed 20/3/2021.

[10] Cricut. 2020. Cricut. https://cricut.com/. Accessed 20/3/2021.

A. Team member's contributions

Explicitly stated contributions of each team member to the final project.

Mikhail Fedorov (20 % of work)

- Coding the main algorithm
- Preparing the GitHub Repo

Vsevolod Avilkin (20 % of work)

- Analyzing code on errors
- Rerunning code with different settings
- Preparing visual materials (figures, tables)

Svetlana Pavlova (20 % of work)

- improved algorithm with gumbel-softmax as a tool for discretization
- wrote sections dedicated to gumbel-softmax activation function

Victor Adamovich(20 % of work)

- responsible for presentation
- main contributor in making a report

Artem Vergazov (20 % of work)

- Code verification and rerunning
- Writing sections 3, 4, 5 of this report

B. Reproducibility checklist

Answer the questions of following reproducibility checklist. If necessary, you may leave a comment.

1. A ready code was used in this project, e.g. for replication project the code from the corresponding paper was used.

   ☐ Yes.
   ☑ No.
   ☐ Not applicable.

   General comment: If the answer is **yes**, students must explicitly clarify to which extent (e.g. which percentage of your code did you write on your own?) and which code was used.

   Students' comment: None

2. A clear description of the mathematical setting, algorithm, and/or model is included in the report.

   ☑ Yes.
   ☐ No.
   ☐ Not applicable.

   Students' comment: None

3. A link to a downloadable source code, with specification of all dependencies, including external libraries is included in the report.

   ☐ Yes.
   ☐ No.
   ☐ Not applicable.

   Students' comment: None

4. A complete description of the data collection process, including sample size, is included in the report.

   ☐ Yes.
   ☐ No.
   ☑ Not applicable.

   Students' comment: None

5. A link to a downloadable version of the dataset or simulation environment is included in the report.

   ☑ Yes.
   ☐ No.
   ☐ Not applicable.

   Students' comment: None

6. An explanation of any data that were excluded, description of any pre-processing step are included in the report.

   ☑ Yes.
   ☐ No.
   ☐ Not applicable.

   Students' comment: None

7. An explanation of how samples were allocated for training, validation and testing is included in the report.

   ☐ Yes.
   ☐ No.
   ☑ Not applicable.

   Students' comment: None

8. The range of hyper-parameters considered, method to select the best hyper-parameter configuration, and specification of all hyper-parameters used to generate results are included in the report.

   ☐ Yes.
   ☑ No.
   ☐ Not applicable.

   Students' comment: None

9. The exact number of evaluation runs is included.

   ☑ Yes.
   ☐ No.
   ☐ Not applicable.

   Students' comment: The number of epochs

10. A description of how experiments have been conducted is included.

    ☐ Yes.
    ☐ No.
    ☑ Not applicable.

    Students' comment: None

11. A clear definition of the specific measure or statistics used to report results is included in the report.

    ☑ Yes.
    ☐ No.
    ☐ Not applicable.

    Students' comment: None

12. Clearly defined error bars are included in the report.

    ☐ Yes.
    ☑ No.
    ☐ Not applicable.

    Students' comment: None

13. A description of the computing infrastructure used is included in the report.

    - [x] Yes.
    - [ ] No.
    - [ ] Not applicable.

    Students' comment: None