Spatial Smoothing with the L_0 Norm to Remove Compression Artifacts in Cartoon Images

Josh Kelle

December 7, 2016

1 Introduction

This project investigates two statistical methods for removing noise from cartoon images caused by lossy compression. Both methods, α-expansion [2] and L_0 gradient minimization [4], frame this spatial smoothing task as an optimization problem.

The α-expansion method models the image as a graph and uses graph cuts to compute a labeling that approximately minimizes graph energy. The L_0 gradient minimization method introduces auxiliary variables into the objective function and alternates solving smaller subproblems.

I describe both methods in moderate detail. I then apply both methods to an example cartoon image and compare results.

2 The Cartoon Domain

Figure 1 shows an example image of a cartoon. The image has been compressed and shows the resulting undesired artifacts. Ideally, the girl's bow would be solid pink with just a single pixel value (aside from the black contour). In figure 1c the two characters exiting the building should be composed of clothing and body parts that are each a constant color value. Instead, the compression introduces a lot of noise in this complex scene.

3 The Objective Function

Let v be the raw, noisy image composed of pixels v_i. Let u be the smoothed image composed of pixels u_i. The objective function f is defined as the sum of two terms.

$$f(u, v) = f_{data}(u, v) + f_{smooth}(u)$$

The f_{data} term measures fit to the data and decreases when u and v are similar. f_{smooth} measures smoothness and decreases when pixels u_i are similar to neighboring pixels u_j. The idea is this: if f_{data} and f_{smooth} are well crafted, minimizing f will result in an optimally de-noised image u.

3.1 Choosing f_{data}

If we assume that observed pixels v_i are IID normally distributed with mean u_i and some variance σ^2, that is $v_i = u_i + e_i$ where $e_i \sim N(0, \sigma^2)$, then the negative log likelihood yields a quadratic term

$$f_{data}(u, v) = \sum_{i=1}^{n} (u_i - v_i)^2 = \|u - v\|_2^2$$

Intuitively, this penalizes u more as it deviates from v. I use this definition of f_{data} in all experiments.

3.2 Choosing f_{smooth}

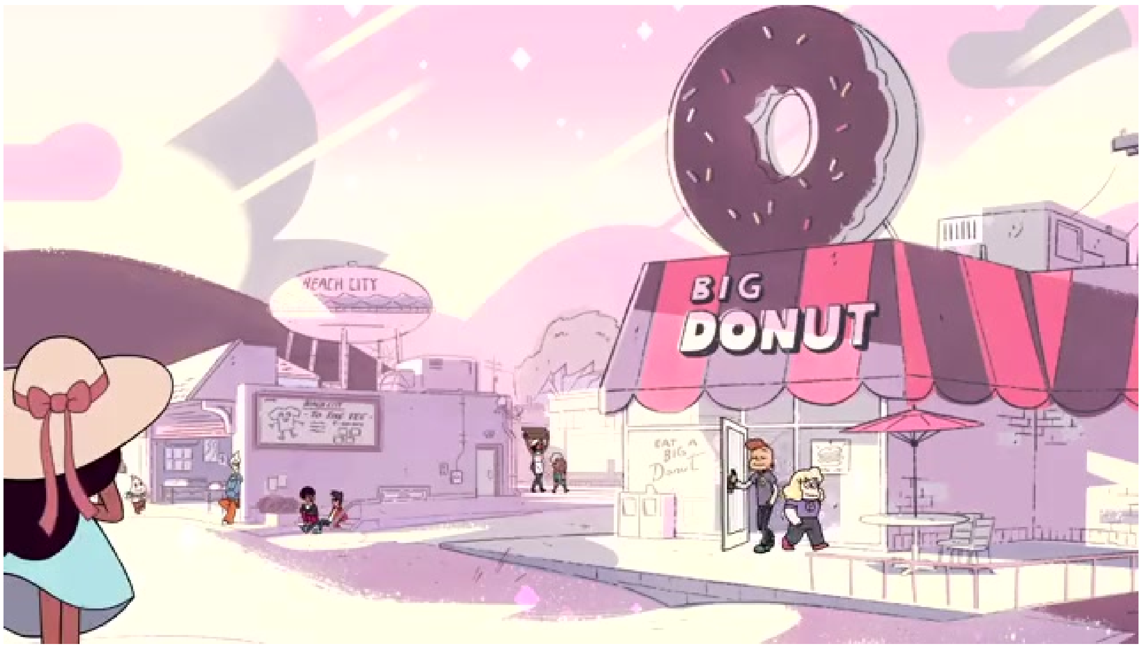
When reasoning about image smoothness, it is often helpful to represent the image as a 4-connected 2D grid graph G where each node u_i represents a pixel. There exists an edge e_{ij} between every adjacent pair of pixels u_i and u_j. This graph structure is illustrated in Figure 2.

Many common smoothing functions are a function of Du where u is the vector representation of the smoothed image and D is the oriented edge matrix. That is, D is constructed such that Du gives a vector of the pairwise differences $u_i - u_j$ for every edge in G. I briefly examine three common definitions of f_{smooth} and compare their usefulness to the task of compression artifact removal of cartoon images.

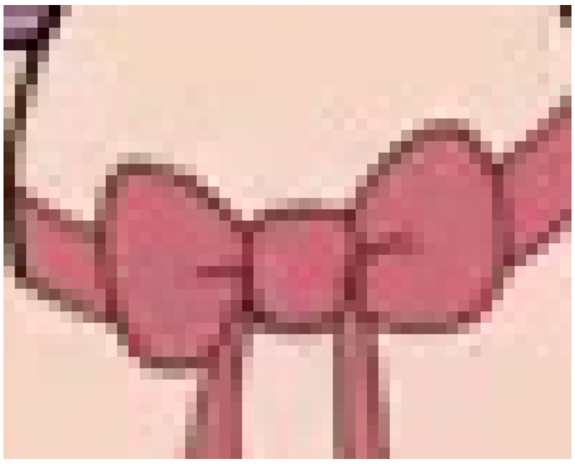
Laplacian smoothing: Laplacian smoothing measures smoothness by the L_2 norm of Du.

$$f_{smooth} = \lambda \|Du\|_2^2$$

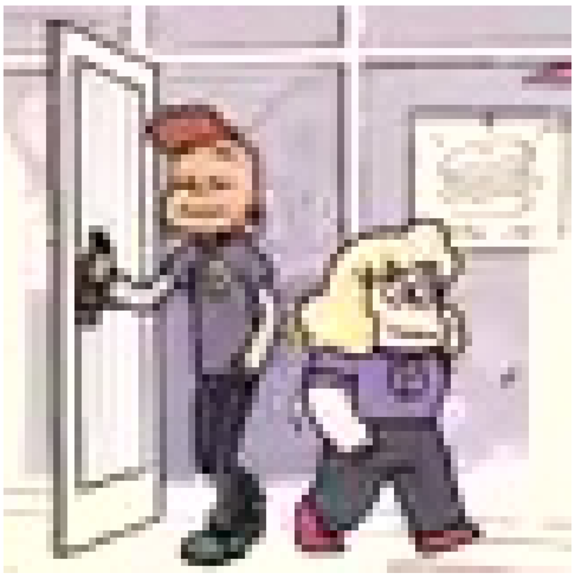
Here λ is simply a constant hyperparameter that governs the balance between fitting the data and producing a smooth image. Using the L_2 norm has the effect of encouraging near-Gaussian smoothness. In this case, optimizing f is approximately equivalent to applying a Gaussian blur convolution to the image. This will



(a) original noisy image



(b) cropped and zoomed-in section of the girl's bow



(c) cropped and zoomed-in section of characters exiting the donut shop

Figure 1: An example cartoon image. Part 1a is the full 640 x 360 resolution image. Parts 1b and 1c are smaller regions in of the image, zoomed in to make compression artifacts easier to see.

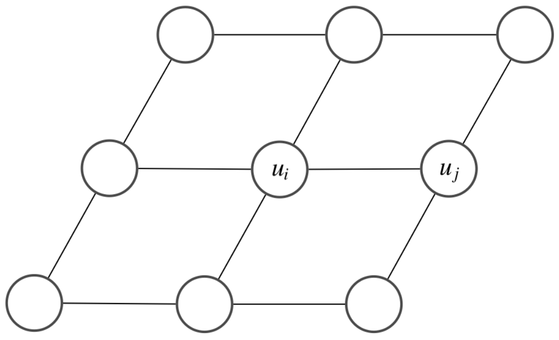


Figure 2: The graphical structure of G. Each pixel u_i of the smoothed image is connected to its four neighbors u_j.

ease the appearance of compression artifacts, but it will also diminish sharp edges. In the context of cartoons, the true image usually consists of several regions of constant color with sharp color differences at their boundaries. Therefore, Laplacian smoothing is not a good choice for this domain.

Graph-Fused LASSO: Graph-fused LASSO (GFL) measures smoothness by the L_1 norm of Du.

$$f_{smooth} = \lambda \left\| Du \right\|_1$$

In contrast to Laplacian smoothing, using the L_1 norm has the effect of encouraging regions of constant color while maintaining sharper changes between regions. However, adjacent regions are still encouraged to be similar in color. For the cartoon dataset, GFL will ease the appearance of compression artifacts, but adjacent regions need not be similar in color. In fact they usually aren't, as shown in Figure 1. Furthermore, using the L_1 also causes the regions' color value of to "shrink" towards the mean color, another undesirable property.

Potts Model: The Potts Model defines the smoothness function as follows

$$f_{smooth} = \lambda \min(1, |u_i - u_j|) = \lambda \left\| Du \right\|_0$$

Since pixels u_i are integers, this is equivalent to counting the number adjacent pixels that differ in color. This definition is equivalent to the L_0 norm of Du. Similar to the GLF, this model encourages regions of constant color. In contrast to the GFL, adjacent color regions incur the same penalty regardless of how similar or different they are in color. This is an appropriate property for the cartoon domain. I use this definition of f_{smooth} in all experiments.

Putting this all together, we have the following objective function:

$$\begin{aligned} f(u, v) &= f_{data}(u, v) + f_{smooth}(u) && (1) \\ &= \left\| u - v \right\|_2^2 + \lambda \left\| Du \right\|_0 && (2) \end{aligned}$$

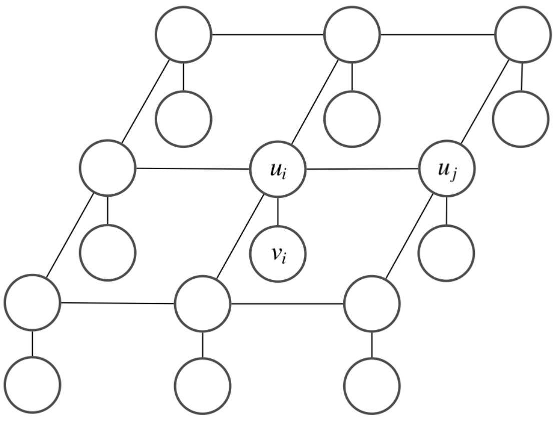


Figure 3: The graphical structure of G'. Each pixel u_i of the smoothed image is connected to its four neighbors u_j in the smoothed image and its corresponding pixel v_i in the raw image.

4 The α-Expansion Method

The α-expansion method published by Boykov et al. (2001) models the image as a graph and uses a series of graph cuts to approximately minimize the energy of the graph [2]. In this section, I define the graph, its energy function, graph cuts on a binary image, and finally the full α-expansion algorithm.

Graph: Define a graph G' to be an augmented version of G. The vertex set contains nodes u_i that represent pixels in the smoothed image and nodes v_i that represent corresponding pixels in the noisy raw image. Like in G, there exists an edge e_{ij} between every adjacent pair of nodes u_i and u_j. Additionally, there exists an edge e_i between every pair of nodes u_i and v_i. Figure 3 illustrates the graph structure.

Energy function: The energy function $E : G' \to \mathbb{R}$ is mathematically identical to the objective function f defined in the previous section.

$$\begin{aligned} E(u, v) &= E_{data}(u, v) + E_{smooth}(u) \\ &= \left\| u - v \right\|_2^2 + \lambda \left\| Du \right\|_0 \end{aligned}$$

Aside: In context of graphs, it is convention to refer to the objective function as an "energy function." Minimizing energy to reach an optimal state is an analogy borrowed from physics.

Graph cuts: Consider the case of a binary image - an image with only two colors. The image is modeled by the graph G' but with two extra nodes s and t. For every node u_i there are two new edges, one to s and one to t. The nodes s and t do not correspond to pixels in the image, but instead correspond to the two colors, or *labels*. An st-cut on the graph is defined as a minimal set of edges e_{ij} such that removing these edges partitions the graph into two connected components, one connected to s and the other to t. That is, once these edges are removed, there is no longer a path from s to t. Nodes u_i with a path to t are labeled with color t, and vise versa for those with a path to s. A min-cut is the st-cut whose sum of edge weights is minimum. A min-cut corresponds to a labeling u that minimizes the energy, and is thus the optimally de-noised image.

α-expansion: The special case of binary labeling can be solved exactly in polynomial time, but the general case of multiple labels remains NP-hard. The α-expansion algorithm approximates the exact solution by applying a series of binary graph cuts, each of which takes a local greedy move toward minimizing the energy. A single expansion move is as follows: for a given label α, let any pixels whose current label is not α either switch to α or remain the same. Given a label α, construct a new graph such that the min-cut labeling minimizes energy.

The α-expansion algorithm iteratively cycles through all labels, applying the single expansion moves until energy stops decreasing. It is important to note that α-expansion is guaranteed to decrease energy only if the E_{smooth} is a metric, meaning it satisfies these three properties:

1. $g(\alpha, \beta) = 0 \leftrightarrow \alpha = \beta$, or $g(\alpha, \beta) \neq 0 \leftrightarrow \alpha \neq \beta$
2. $g(\alpha, \beta) = g(\beta, \alpha) \geq 0$
3. $g(\alpha, \beta) \leq g(\alpha, \gamma) + g(\gamma, \beta)$

5 The L_0 Gradient Minimization Method

Xu et al. (2011) describe a different method for approximately minimizing equation 2 in their paper Image Smoothing via L_0 Gradient Minimization [4]. Instead of modeling the image as a graph and computing graph cuts, the authors augment the objective function with auxiliary variables and propose an alternating minimization solver that iteratively updates different sets of variables until convergence. This method has the same effect of suppressing low-amplitude image structures while preserving salient edges. In addition to compression artifact removal in cartoons, the authors also claim their method is useful for edge extraction, non-photorealistic effect generation, and detail enhancement.

The authors formulate the objective function as

$$\min_S \left\{ \sum_p (S_p - I_p)^2 + \lambda C(S) \right\} \tag{3}$$

where I is the noisy image, S is the smoothed image, p indexes pixels, λ is the smoothness hyperparameter as defined before, and $C(S) = \#\{p \mid |\partial_x S_p| + |\partial_y S_p| \neq 0\}$ counts the number of times a pixel S_p differs from its neighbors. It is easy to see that equation 3 is mathematically equivalent to equation 2.

The authors approximately solve this optimization problem by introducing auxiliary variables h_p and v_p into the objective function to create a new augmented objective

$$\min_{S,h,v} \left\{ \sum_p (S_p - I_p)^2 + \lambda C(h, v) + \sum_p \beta((\partial_x S_p - h_p)^2 + (\partial_y S_p - v_p)^2) \right\}$$

where $C(h, v) = \#\{p \mid |h_p| + |v_p| \neq 0\}$ and β is an automatically adapting parameter that controls the similarity between (h, v) and their respective gradients. Notice that this equation approaches equation 3 as β becomes large. The variables h_p and v_p correspond to $\partial_x S_p$ and $\partial_y S_p$ respectively (h for "horizontal" and v for "vertical").

Each iteration of the solver solves two subproblems. The first subproblem updates S while keeping h and v fixed. The second subproblem updates h and v while keeping S fixed.

Updating S: Omitting terms that don't involve S gives the following quadratic optimization subproblem, which can be solved by gradient descent.

$$\min_S \left\{ \sum_p (S_p - I_p)^2 + \sum_p \beta((\partial_x S_p - h_p)^2 + (\partial_y S_p - v_p)^2) \right\}$$

The authors describe an alternative, faster method of solving this subproblem using the Fast Fourier Transform, but I won't cover it here.

Updating (h, v): Omitting terms that don't involve h or v gives the following optimization problem.

$$\min_{h,v} \left\{ \frac{\lambda}{\beta} C(h, v) + \sum_p ((\partial_x S_p - h_p)^2 + (\partial_y S_p - v_p)^2) \right\}$$

The key here is this expression can be decomposed such that each h_p and v_p are estimated individually, which gives the following:

$$\sum_p \min_{h_p, v_p} \left\{ \frac{\lambda}{\beta} H(|h_p| + |v_p|) \right. \\ \left. + (\partial_x S_p - h_p)^2 + (\partial_y S_p - v_p)^2 \right\}$$

where $H(|h_p| + |v_p|)$ is a binary indicator function that returns 1 if $|h_p| + |v_p| = 0$ and 0 otherwise. Each term in the summation reaches a minimum under the condition

$$(h_p, v_p) = \begin{cases} (0,0) & (\partial_x S_p)^2 + (\partial_y S_p)^2 < \lambda/\beta \\ (\partial_x S_p, \partial_y S_p) & \text{otherwise} \end{cases}$$

The authors provide a proof of this claim in their paper, but I won't repeat it here.

6 Experiments

In this section I apply both methods to the cartoon shown in figure 1. In both cases, I use code provided from the respective authors.

6.1 Smoothing with α-expansion

Let n be the number of pixels in the image, and let $\mathcal{L} = \{l_1, l_2, ..., l_m\}$ be the set of allowable labels/colors. In the provided C++ code [2, 3, 1], the user defines $f_{data}(u_i, v_i)$ as a matrix $A \in \mathbb{R}^{n \times m}$ where $A_{i,j}$ contains the cost incurred from labeling pixel u_i with color l_j. The user defines $f_{smooth}(u_i, u_j)$ as a matrix $B \in \mathbb{R}^{m \times m}$ where $B_{i,j}$ contains the cost incurred from labeling some pixel with color l_i when one of its neighboring pixels is labeled with color l_j. These two matrices are provided to the solver and fully define the problem.

Note the flexibility of this setup. It allows for any custom definition of f_{data} and f_{smooth}, not just those given by equation 2. However, I define the matrices according to equation 2 unless otherwise specified.

Applying α-expansion to a grayscale image is straightforward. But how can one apply this method to a color image? I explore 3 options here.

Option 1: Independent channels One option is to treat each of the three color channels separately by running three instances of α-expansion to get three smoothed images, and then merge the results. However, these three color channel images might result in a different segmentation, in which case it is unclear to me how to merge them into a single smoothed color image.

Option 2: Color encoding Another option is to map a color $c = (c_1, c_2, c_3) \in \mathbb{R}^3$ to a scalar $c' \in \mathbb{R}$. Since each color channel has the range $[0, 255]$, I construct the following mapping that simply concatenates the 8 bits of each color channel: $c' = (c_1 \ll 16) \vee (c_2 \ll 8) \vee c_3$ where "$\ll$" is the bit-wise left shifting operator and "$\vee$" is the bit-wise OR operator. Given c', one can recover the three separate channels by similar bit-wise logic.

With this mapping, the size of the label space $\mathcal{L}$ is $m = 256^3 = 16,777,216$. The image in figure 1a has dimension 640×360. This means the data cost matrix A would have dimension $n \times m = 230,400 \times 16,777,216$ which is more than 3.8 trillion entries. This is intractable.

Instead of using all possible 256^3 labels, what if we used just the labels present in the original image? The worst case occurs when every pixel in the image is a different color. This leads to $m = n = 640 * 360 = 230,400$. Then A would have $n^2 = 230,400^2 \approx 53$ billion entries. Assuming each element takes 32 bits of memory, this results in a matrix of size about 200 GB. This is still intractable.

But that's worst case. The image in figure 1a actually has only 39,725 distinct colors. This would result in a matrix A with $m \times n = 39,725 * 230,400 \approx 9$ billion entries, which would require about 34 GB of memory. Still intractable for my laptop.

Luckily, the authors' code has an alternative to instantiating these large matrices. Instead, one can define C++ functions that compute these costs on the fly, as needed. (If memory is not a concern, this option will probably slower than using the matrices.) I try this option, but the algorithm still needs to cycle through all m labels, and in my experience each iteration can take several minutes. So it is still very slow.

Option 3: Grayscale transfer One option is to convert the color image to grayscale, invoke α-expansion to segment the image into color-constant regions, convert back to a color image, and paint these regions with the their respective average color in the original image. By "paint" I mean to overwrite pixel values. In this scenario, there are only $m = 256$ labels. This results in the data cost matrix A having $256 * 230,400 \approx 60$ million, which requires about 225 MB of memory. This is tractable! The smoothness cost matrix B will have $256^2 = 65,536$ entries which turns out to require 256 KB of memory. Not much at all compared to A.

Therefore, I chose to pursue option 3, grayscale transfer. Results across different values of lambda are shown in figure 5. For smaller values of λ, you can see how the sky gets merged into fewer regions. As λ gets larger, the image becomes smoother. For $\lambda = 1600$, the image is

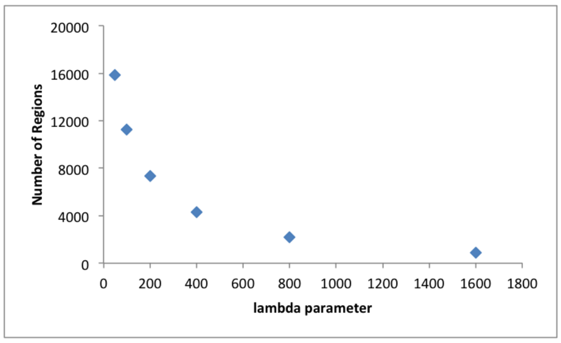


Figure 4: This plot shows how the number of regions decreases as λ increases. In other words, the image gets segmented into fewer colors, becoming smoother.

over-smoothed. Much of the salient detail has been erased. Figure 4 shows how the number of regions varies as a function of λ. Here, "number of regions" is a proxy for smoothness - fewer regions implies more smooth. Figures 6 and 7 shows results for the zoomed-in sections of the girls bow and the characters exiting the donut shop.

6.2 Smoothing with L_0 gradient minimization

The authors of [4] provide MATLAB code that implements their solver. It handles color images out of the box, so using it is more straightforward than using the provided α-expansion code. I did, however, slightly modify the MATLAB code - I increased the parameter β_{max} which increases accuracy at the cost of longer execution time. Results for different values of λ are shown in figures 8 - 10.

Notice how the results from this method also become more smooth as λ increases. However, the results aren't the same as those from α-expansion. The segmentation and smoothing style is different between the two methods despite having mathematically equivalent objective functions. This is because neither method solves the optimization problem exactly. Both methods are approximations, and can thus lead to different results.

7 Comparing methods

It is clear that both methods can be used to remove compression artifacts in images. So which method should be used? This section briefly reviews the tradeoffs between the two methods.

Style: Figures 5 and 8 show that the two methods produce different high-level styles when enough smoothing happens (when λ is large). Results from L_0 gradient minimization produce smoother, blurrier large structures. An example of this is the sky in these figures.

Time vs flexibility: In my experience, L_0 gradient minimization is several times faster than α-expansion. Part of the reason is that the L_0 gradient minimization method always assumes the objective function to be defined as in equation 2. The solver leverages this fact to produce a faster algorithm. On the other hand, with α-expansion the practitioner is free to define any arbitrary data cost function f_{data} and smoothness cost function f_{smooth} they want (as long as f_{smooth} satisfies the three metric properties). This is powerful and appealing to anyone who wants to experiment with smoothness cost functions. The price paid for this flexibility is longer computation time.

In short, neither method is superior in all cases. The right method to use depends on the goals and constraints of the project.

8 Conclusion

I investigated two methods for removing lossy compression artifacts from cartoon images. Both methods aim to achieve this goal by minimizing the objective function defined by equation 2. However, minimizing this objective exactly is NP-hard, so both methods compute an approximation. One method, α-expansion, models the image as a graph and computes a low-energy labeling via graph cuts. The other method, L_0 gradient minimization, augments the the objective function with auxiliary variables and alternates solving for one set of variables while keeping the rest fixed. I showed that both methods can successfully smooth out compression artifacts, though doing so without over smoothing requires careful tuning of the λ parameter.

References

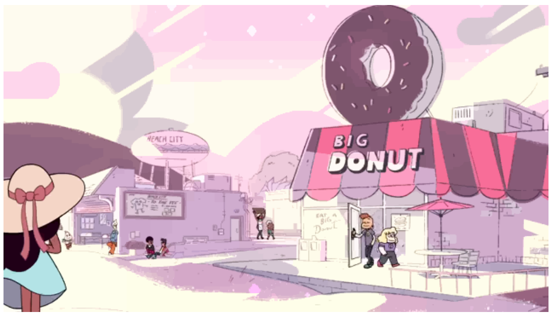
[1] Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE transactions on pattern analysis and machine intelligence*, 26(9):1124–1137, 2004.

[2] Y. Boykov, O. Veksler, and R. Zabih. Fast approximate energy minimization via graph cuts. *IEEE Transactions on pattern analysis and machine intelligence*, 23(11):1222–1239, 2001.

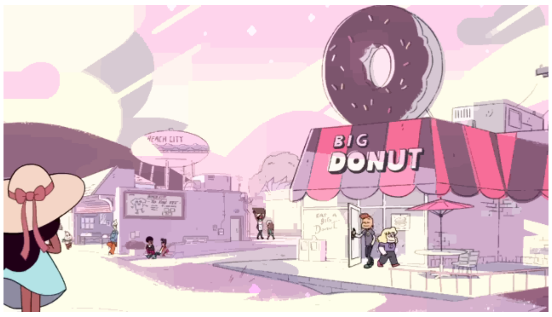
[3] V. Kolmogorov and R. Zabin. What energy functions can be minimized via graph cuts? *IEEE transactions on pattern analysis and machine intelligence*, 26(2):147–159, 2004.

[4] L. Xu, C. Lu, Y. Xu, and J. Jia. Image smoothing via l 0 gradient minimization. In *ACM*

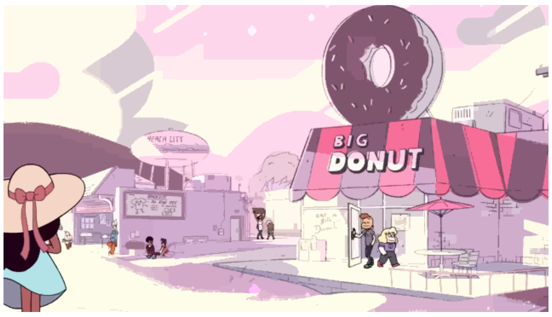
Transactions on Graphics (TOG), volume 30, page 174. ACM, 2011.



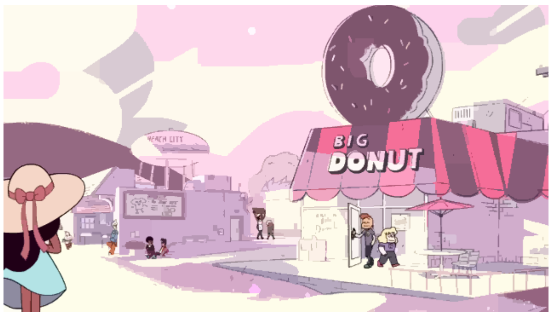
(a) $\lambda = 50$



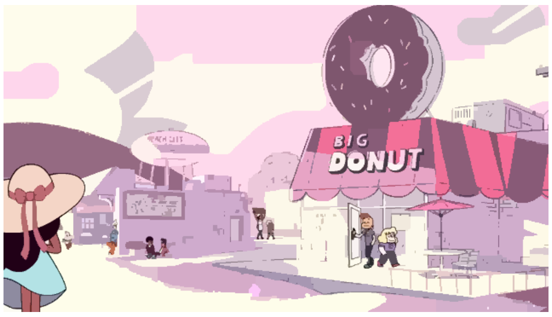
(b) $\lambda = 100$



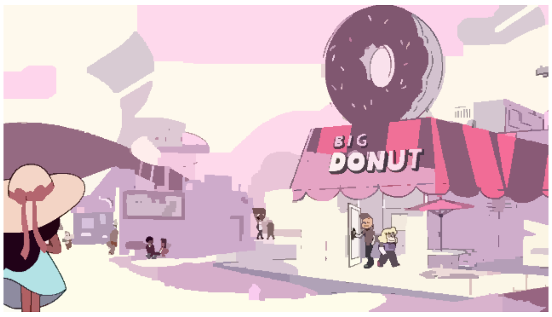
(c) $\lambda = 200$



(d) $\lambda = 400$

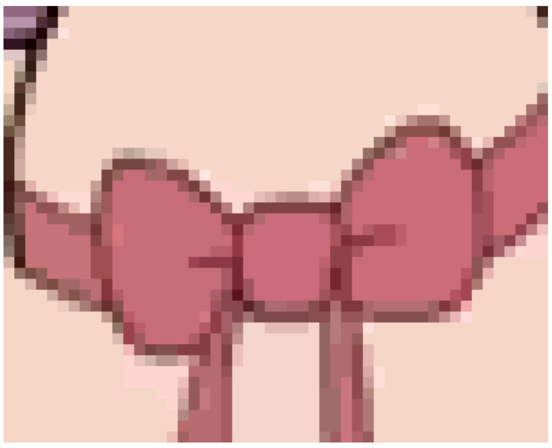


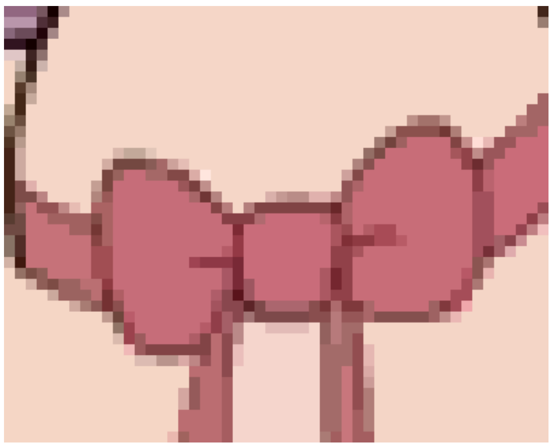
(e) $\lambda = 800$

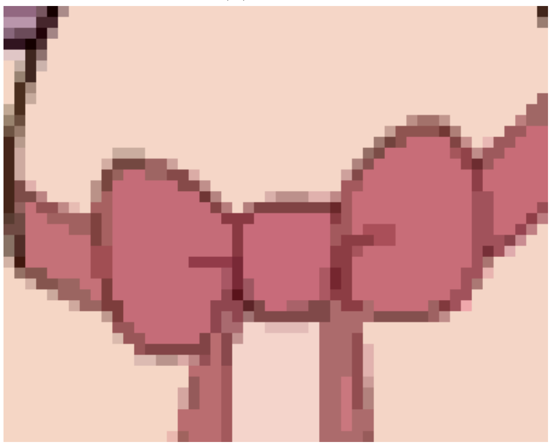


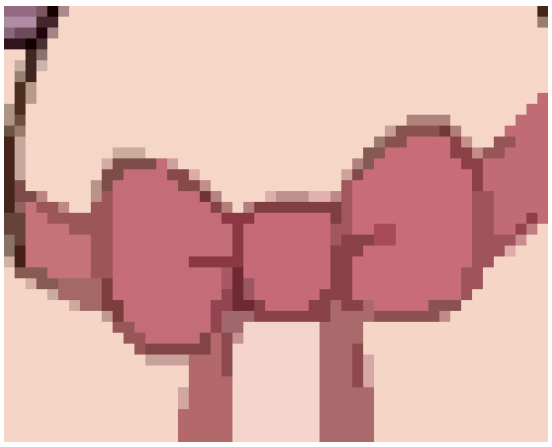
(f) $\lambda = 1600$

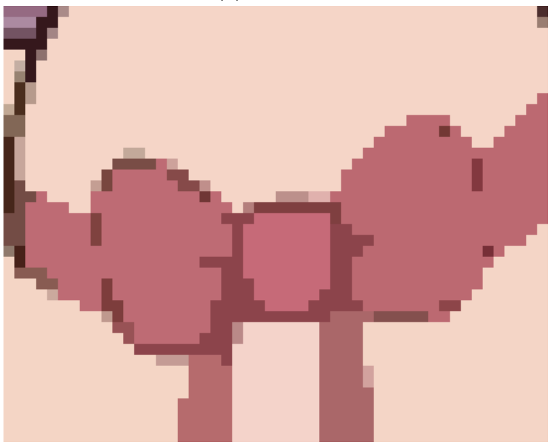
Figure 5: Results of α-expansion over different values of λ. These results were generated by the grayscale transfer method.


(a) $\lambda = 50$


(b) $\lambda = 100$


(c) $\lambda = 200$


(d) $\lambda = 400$


(e) $\lambda = 800$

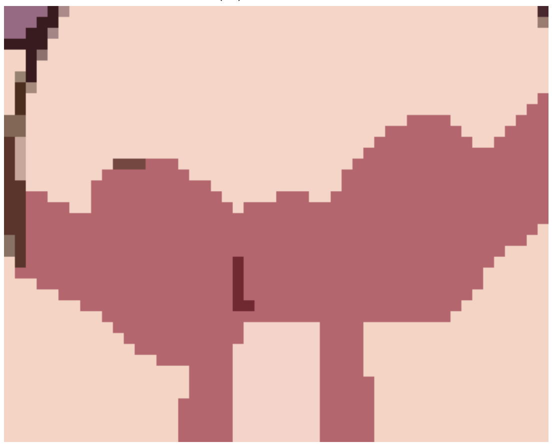
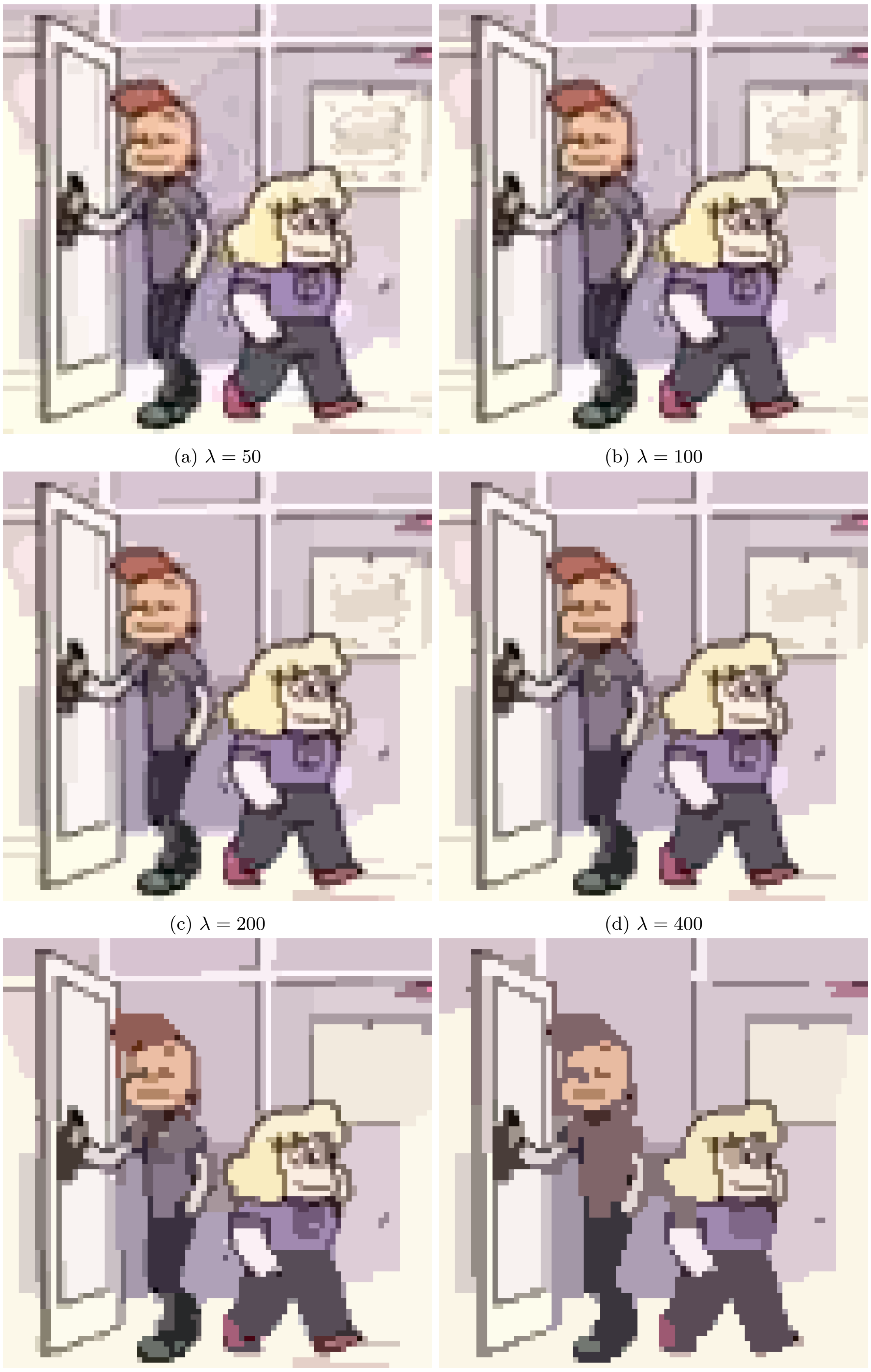

(f) $\lambda = 1600$

Figure 6: Results of α-expansion over different values of λ, zoomed in to see results more easily. These results were generated by the grayscale transfer method.

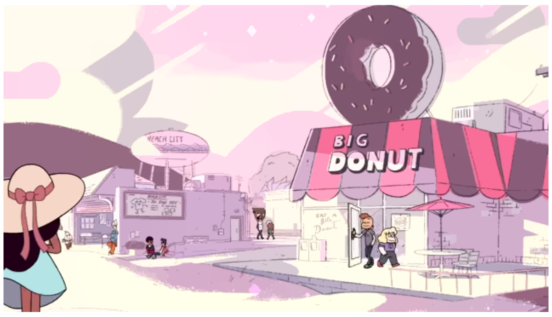


(a) $\lambda = 50$ (b) $\lambda = 100$

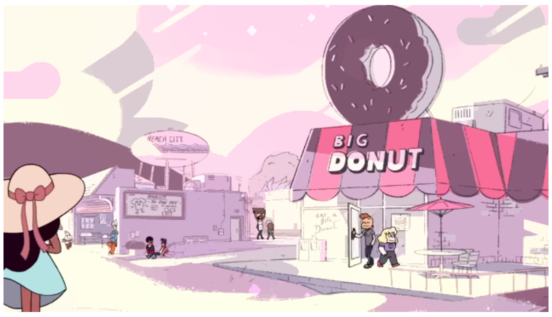
(c) $\lambda = 200$ (d) $\lambda = 400$

(e) $\lambda = 800$ (f) $\lambda = 1600$

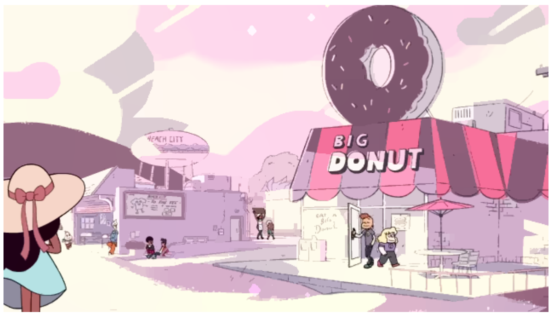
Figure 7: Results of α-expansion over different values of λ, zoomed in to see results more easily. These results were generated by the grayscale transfer method.



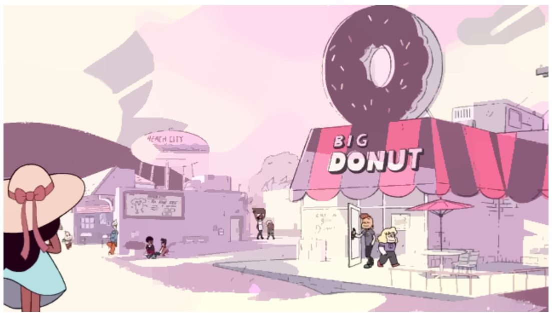
(a) $\lambda = 0.001$



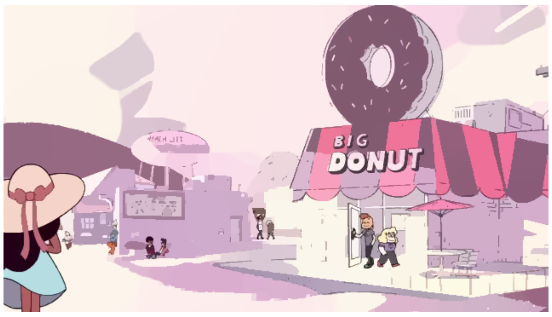
(b) $\lambda = 0.002$



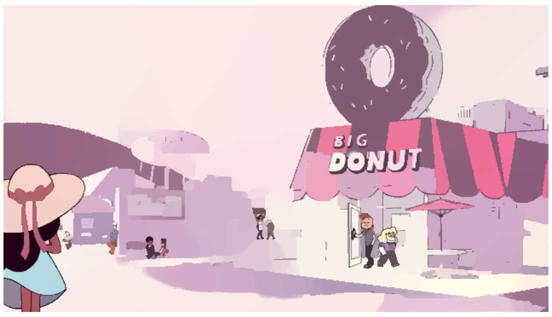
(c) $\lambda = 0.004$



(d) $\lambda = 0.008$

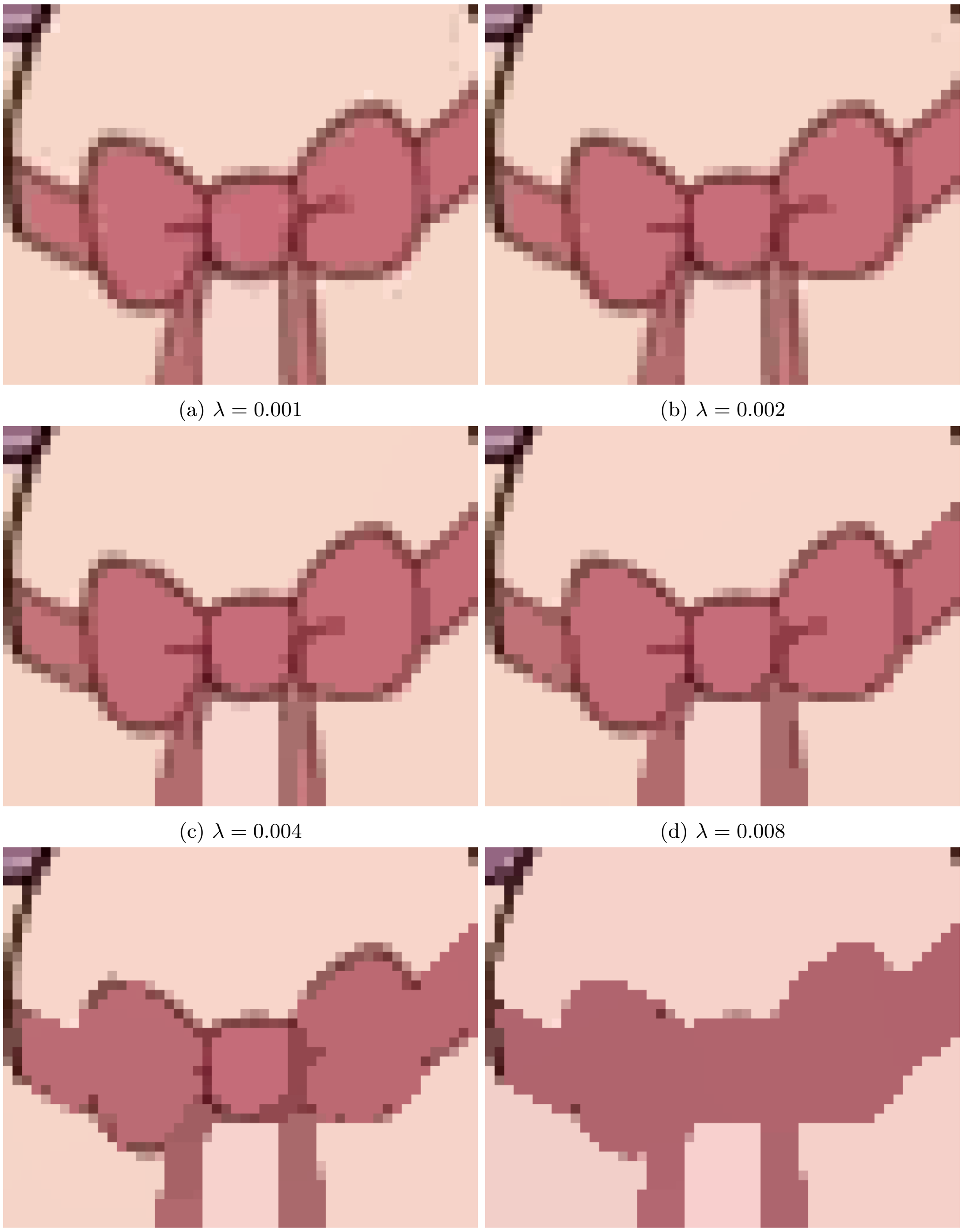


(e) $\lambda = 0.016$



(f) $\lambda = 0.032$

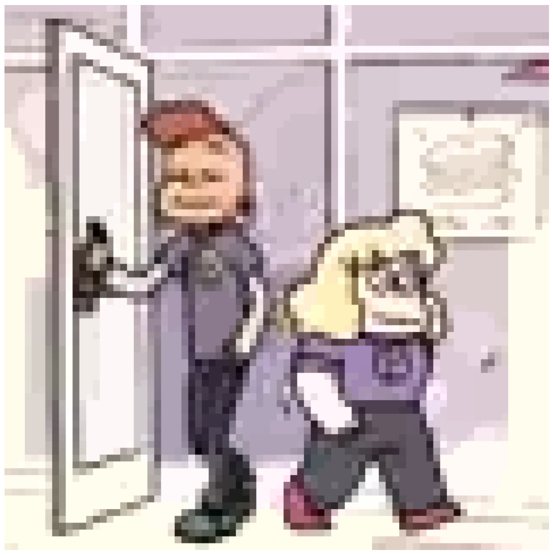
Figure 8: Results of L_0 gradient minimization over different values of λ.

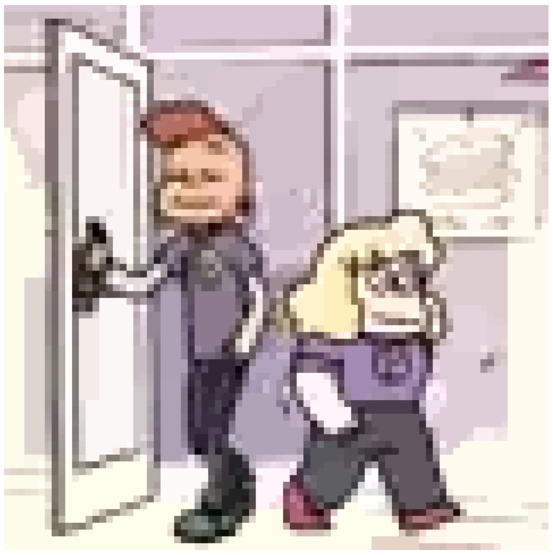

(a) $\lambda = 0.001$ (b) $\lambda = 0.002$

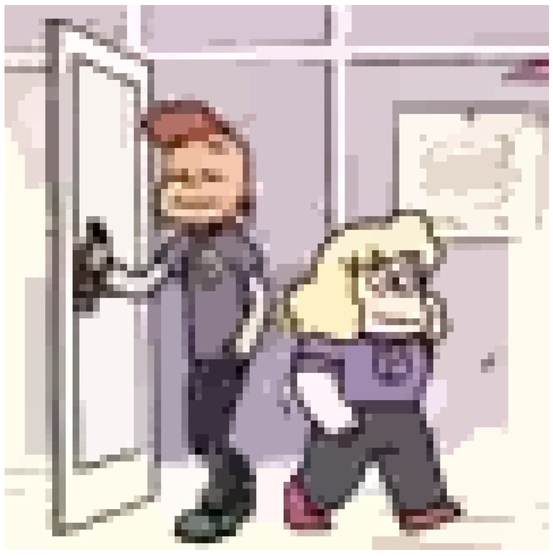
(c) $\lambda = 0.004$ (d) $\lambda = 0.008$

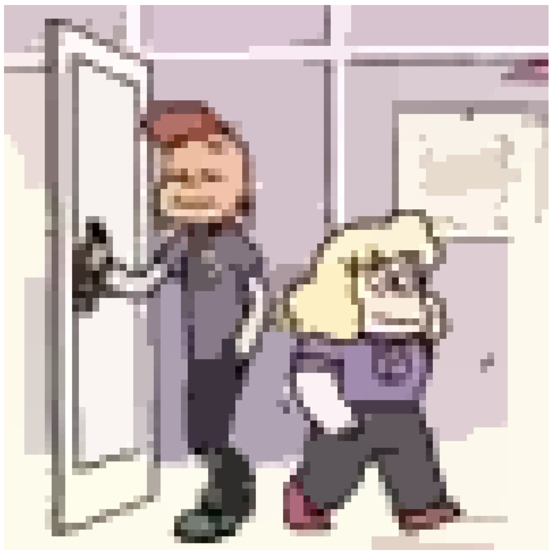
(e) $\lambda = 0.016$ (f) $\lambda = 0.032$

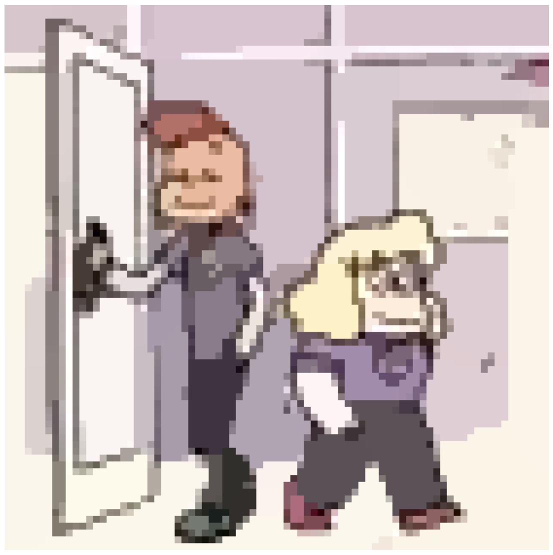
Figure 9: Results of L_0 gradient minimization over different values of λ, zoomed in to see results more easily.


(a) $\lambda = 0.001$


(b) $\lambda = 0.002$


(c) $\lambda = 0.004$


(d) $\lambda = 0.008$


(e) $\lambda = 0.016$

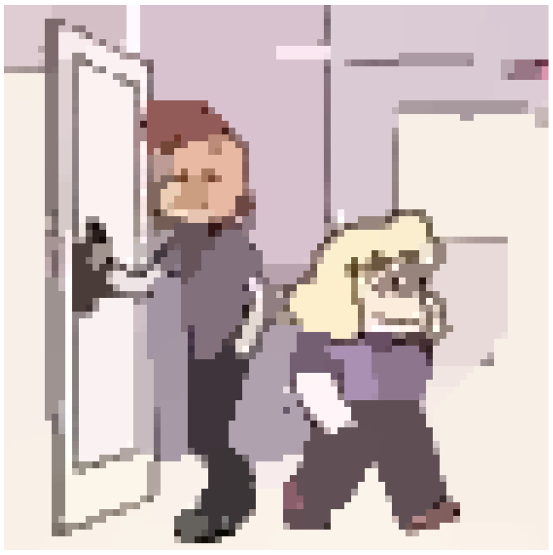

(f) $\lambda = 0.032$

Figure 10: Results of L_0 gradient minimization over different values of λ, zoomed in to see results more easily.