Firelight’s Inner Workings

Introduction
Understanding Images
Initial Approach (K-means)
Problems with K-means
Quantization with Principal Component Analysis
Intuitive Derivation
Formalizing the Calculations
PCA: Proof of Maximizing Variance
Using PCA to split our data into colors
Dynamically choosing the number of clusters K

Introduction

The goal of Firelight is provide users with an ambilight solution to sync up any programmable lights to the contents of their computer screen. The process for doing this can be split into 3 parts:

1. Capture the contents (pixels) of the screen
2. From the contents, determine a dominant color
3. Send this color to the light system of the user’s choice

The contents of this document will capture the solution to the second (2) of these parts: how can we effectively transform the contents of the captured screen into a color that *feels* like it matches what the user is watching?

Understanding Images

At each time step, we capture a screenshot of the image. A screenshot is comprised of individual pictures. This means that we can think of our screenshot as an $M \times N$ matrix of pixels. Each pixel furthermore has three values associated with it: Red, Green, and Blue (RGB). With this in mind, our image is really just an $M \times N \times 3$ matrix of values, ranging from 0 to 255 (the ranging of RGB values).

One more question to ask is: do we care about the location of these pixels in relation to each other? Does their position on the screen matter?

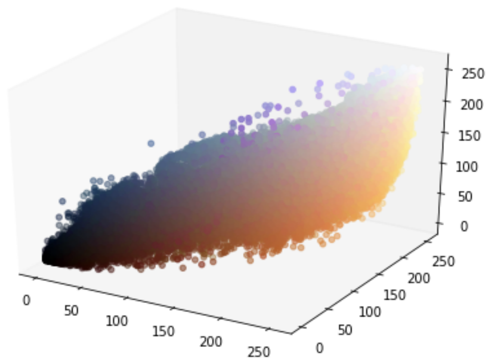
Since the product only aims to display a single color (for now), Firelight should aim to capture the *feel* of the image - the atmosphere it presents. Therefore, we choose to ignore the location of each pixel individually since we want to capture only one color from all the pixels as a whole.

Following this logic, let's flatten first two dimensions of the the matrix (squeeze the M and R dimensions together). This leaves us with an $MN \times 3$ matrix.

Initial Approach (K-means)

Knowing that the pixels in an image, ignoring location, can be represented by an $MN \times 3$ matrix, we can take a new perspective. This data can also be thought of as an MN *list* of coordinates in $\mathbb{R}^3$, where the axes represent how much red, green, and blue is in each pixel (data point).

For example, let's take the $240px \times 427px$ image below (left). For each pixel in the image, we will plot it in $\mathbb{R}^3$ as (R, G, B). The result is the graph shown below (right). For visualization purposes, each pixel is colored as its original value.



With the insight that the pixels are really just coordinates, we can utilize **k-means clustering**. K-means clustering is an iterative algorithm which attempts to group N data points into K clusters, where each data point x is assigned to only one cluster. Data points are assigned to the cluster centroid (or cluster mean) that they are closest too.

Formally, the objective function to be minimized can be expressed as

$$\min \sum_{k} \sum_{x \in S_k} ||x_n - \mu_k||^2$$

where x represents a data point, S_k is the set of data points in cluster k, and μ_k is the centroid (mean) of S_k.

💡 Briefly summarized, the **k-means clustering algorithm** is:

1. Initialize cluster centroids μ_k randomly
2. Until convergence:
    a. Assign data point x_i to the closest cluster (determined by the $L2$ norm)
    b. For every cluster, recalculate the cluster centroids using the updated cluster assignments

By using k-means clustering, we attempt to *quantize* the colors in the target image. In other words, we are attempting to reduce the number of distinct colors in the image to a select few that are similar to the original image. K-means will group coordinates that are close together in the RGB color space into clusters. The centroids of these clusters are the average of all the coordinates that are assigned to that cluster. Therefore, each centroid is a general *approximation* of all the individual colors in its corresponding cluster.

With k-means, we now have a way to obtain K candidate colors to represent the image. Now, we must narrow it down. To choose just one, we use a combination:

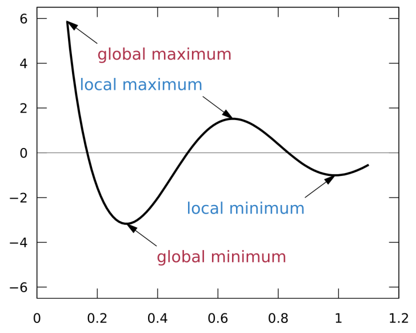
1. The size of cluster that the the centroid corresponds to
2. The “colorfulness” the candidate color (we want interesting colors, e.g. not just black).

For example, let's say we run k-means to obtain $K = 12$ different candidate colors (μ_k in the objective function above). Each color μ_k represents the average of the $|S_k|$ colors assigned to its cluster $|S_k|$. It's possible that many of these clusters only have a few data points assigned to them. Since we don't want to choose a color that may only show up in a few pixels in the image, lets keep the centroids corresponding to the 5 largest clusters and discard the rest. Out of these 5 colors, we choose the most "colorful" one, where "colorfulness" is a heuristic defined as the product of brightness and luminance. This is just the current approach, but a different combination of these may be used for a better approach in the future (requires more fine-tuning which I plan to work on).

Problems with K-means

You may be thinking: "This seems pretty arbitrary - Why are we using K=12? What if there are more than 12 noticeably different color in the image? Or less than 12?". And you'd be right to wonder this. This is a limitation of this approach that stems from k-means. Since the number of clusters K is an input to the k-means algorithm, this is a value that we have to provide. Typically in machine learning, K is a hyperparameter that is chosen through hyperparameter tuning (e.g. a grid search involving looping over a range of k_i and choosing whichever one gives the best results on the validation set).

There is also one more problem that surfaces from using this algorithm. **K-means clustering suffers from random initialization**. To understand why, let's imagine the objective function as a graph. The topology of this graph has hills and dips and saddle points. Remember that we are trying to minimize the objective function - we want to be as "low" as possible. With some unlucky initialization, we may start in an area that is a "valley". If we follow the direction opposite of the derivative of this surface, we may get stuck at the bottom of this valley since all directions around this point seem to be increasing in value (worse, since we're trying to *minimize*). This is a **local minimum**. But what if there's a lower point somewhere else that we just can't reach? That is the



(Maxima and Minima)

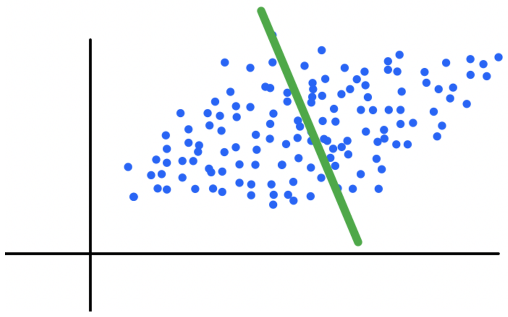
true **global minimum** a.k.a. the optimal solution to the objective function.

Quantization with Principal Component Analysis

To address the problem encountered when using K-Means, we will utilize a common method used in Machine Learning known as **Principal Component Analysis** (PCA) to perform **Hierarchical Quantization**. Before defining these terms, let's establish an assumption along with a way to exploit the assumption to achieve our goal.

Intuitive Derivation

Let's visualize our RGB values as data points on a graph again. If we want to separate (and group) these data points into 2 different colors, how would we do that? Intuitively, we might choose a plane to "cut" the data in half - group all the data points on one side of the plane as a single color, and all the data points on the other side to be the other color.



A 2D example of this is shown in the image above, where the blue dots represent data samples and the green line represents the separating hyperplane. Notice that the hyperplane was chosen such that it separates the data based on the "widest" axis of the data, because we expect different colors to be further away from each other.

What if we want more than 2 colors? For example, what if we want 3 colors? We can use recursion to split one our "colors" further into two more colors, giving us a total of 3 colors. How do we choose which color to split? Following our intuition once again that different colors are further away from each other, we just split the "widest" group of points. In this way, we can continue splitting clusters of points recursively until we get our desired K number of colors. The resulting operation ends up looking like a binary tree, which is why this method is called **hierarchical quantization**.

Formalizing the Calculations

When we choose the “widest” axis of a cluster of points, what we are really doing is choosing the direction which maximizes *variance.* If we want a mathematical way to do this, PCA tells us exactly how.

💡 **Principal Component Analysis (PCA)** tells us that the direction that maximizes variance is the principal component of the covariance matrix X^TX (i.e. the eigenvector of the data corresponding to the largest eigenvalue).

PCA is an extremely common algorithm in unsupervised machine learning, often used for dimensionality reduction to identify important features and drop features which carry less information. If you are familiar with PCA, feel free to skip the next subsection. Otherwise, reading through the derivation will give a better understanding of how PCA can help us.

PCA: Proof of Maximizing Variance

To see this, remember that the Variance is defined as $Var(X) = E[(X-\mu)^2]$.

Lets assume that our dataset X is already centered (i.e. $\mu = 0$). We are trying to find the unit vector w along which the projection of our data X is maximized. The problem can be stated as:

$$\underset{w:||w||_2=1}{\arg\max} \frac{1}{N}||Xw||_2^2$$

From the objective function, we can expand:

$$\frac{1}{N}||Xw||_2^2 = \frac{1}{N}w^TX^TXw$$

Spectral Decomposition tells us: $X^TX = Q\Lambda Q^T$, where Q is an orthonormal matrix of eigenvectors and D is a diagonal matrix of corresponding eigenvalues.

$$\frac{1}{N}||Xw||_2^2 = \frac{1}{N}w^TQ\Lambda^TQ^TQ\Lambda Q^Tw = \frac{1}{N}w^TQ\Lambda^T\Lambda Q^Tw$$

Let’s define $z = Q^T w$. Notice that $||z||_2 = 1$. Now, the problem statement can be rewritten as

$$\underset{w:||w||_2=1}{\arg\max} \frac{1}{N}||Xw||_2^2 = \underset{z:||z||_2=1}{\arg\max} \frac{1}{N} z^T \Lambda z$$

$$= \underset{z:||z||_2=1}{\arg\max} \sum_i \lambda_i z_i^2$$

To maximize this, we want the $z_i = 1$ for the largest λ_i (all the weighting should be on the biggest eigenvalue). So assuming the λ_i are ordered such that λ_0 is the largest eigenvalue, $z = [1, 0, 0, ..., 0]$. Remember that $z = Q^T w$, where Q is the orthonormal matrix of eigenvectors of $X^T X$. Thus, $w = Qz = q_0$, the eigenvector corresponding to the largest eigenvalue. The largest eigenvector is also known as the **principal component**.

Using PCA to split our data into colors

Let q be the principal component of our dataset. We can visualize the principal component q as a normal vector to the hyperplane that is splitting our dataset X. This hyperplane will split the data based on the mean $\mu = \frac{1}{N} \sum x_i$. Since we only care about where the data is located with respect to the direction of the principal component, we can classify our data based on the vector projection Xq. The threshold value is defined as μq. We can then split into two new data sets:

$$X_1 = \{x_i \in X : x_i q <= \mu q\}$$

$$X_2 = \{x_i \in X : x_i q > \mu q\}$$

We can continue recursively splitting these datasets until we get the desired K number groupings. From there, we can take the mean of the data points in each grouping as a color. This RGB value can either be used as the initialization for k-means or as a color value directly (currently, Firelight uses it as the initialization for k-means, but since there is no obvious advantage to doing so, I will likely switch the program to directly use the RGB values calculated from this method).

Dynamically choosing the number of clusters K

Knowing that the eigenvalue measures the variance along its corresponding eigenvector, we can try a new idea. Recall that one of the issues with K-means clustering is that the number of clusters K is a hyperparameter that must be chosen beforehand. This means that for a given image, the algorithm will try finding K clusters when in fact there could be more or less colors.

Using the lessons from PCA, we can attempt to find an optimal stopping criterion for the Hierarchical Quantization algorithm using the maximum eigenvalue. At each recursive layer of the algorithm, check if any of the eigenvalues corresponding the the principal components of the color clusters are above some pre-determined eigenvalue threshold. If none are, then we can stop recursing and splitting clusters, and use the clusters that we have ended up with. Otherwise, continue splitting the cluster with the largest variance (eigenvalue).

💡 **Implementation Note**: Using a Max Heap proves useful here. Clusters can be maintained in the Max Heap according to their corresponding eigenvalue. Each each step of recursion, pop a cluster from the heap and check against the eigenvalue threshold. If the cluster does not meet this stopping criterion, split it into two new clusters and insert both of them into the heap.

This pre-determined eigenvalue threshold can be chosen experimentally. Using a set of test images, run the quantization algorithm using a range of different values. Compare the number of colors returned (along with their values) to your expectations as a human. I found that 3000 worked well as a threshold, but this was not rigorously trialed and should be subject to more extensive fine-tuning.