Using Gaussian Mixture Models with Deep Neural Network features for Image Segmentation.

Victor Megir

Diploma Thesis

Supervisor: Prof. Christophoros Nikou

Ioannina, August 2020

Τμήμα Μηχ. Η/Υ & Πληροφορικής
Πανεπιστήμιο Ιωαννίνων
Department of Computer Science & Engineering
University of Ioannina

Acknowledgments

This work would not have been possible without the consulting and the annotated images provided by **George Sfikas**, Research Associate and Visiting Lecturer on the Department of Computer Science and Engineering of University of Ioannina.

Date: 28/8/2020

Author: Victor Megir

Abstract

Semantic Image Segmentation is a challenging problem in computer vision. Getting good results requires image features that hold semantic, real world information. Recent developments in computer vision however, especially the use of Deep Convolutional Neural Networks have allowed for the extraction of high quality features. These features are the result of state of the art neural networks that have trained on massive datasets. Leveraging these high quality features can allow for quality image segmentation, using simpler techniques and significantly smaller datasets. Statistical modeling and clustering techniques, such as Gaussian Mixture Models can be applied on the feature data, while feature engineering and dimensionality reduction can come a long way in improving the segmentation results.

Keywords: Deep Features, Gaussian Mixture Model, Dimensionality Reduction.

Περίληψη (extended abstract in Greek)

Η κατάτμηση εικόνας αποτελεί ένα από τα πλέον απαιτητικά προβλήματα στον τομέα της υπολογιστικής όρασης, όμως ένα με ενδιαφέρουσες εφαρμογές και μεγάλο ερευνητικό ενδιαφέρον. Η σημαντική πρόοδος που έχει επιτευχθεί τα τελευταία χρόνια οφείλεται στη χρήση μοντέρνων μεθόδων αναγνώρησης προτύπων και ιδιαίτερα τα βαθιά συνελικτικά νευρωνικά δίκτυα. Χάρη στις αρχιτεκτονικές τους, αυτά τα νευρωνικά δίκτυα έχουν τη δυνατότητανα να εξάγουν χαρακτηριστικά από εικόνες, με τα οποία επιτυγχάνουν ασύγκριτα καλύτερα αποτελέσματα σε σχέση με χαρακτηριστικά δομημένα από ανθρώπους.

Η ποιότητα των χαρακτηριστικών που χρησιμοποιούνται, αποτελούν κρίσιμο παράγοντα για την ποιότητα της κατάτμησης εικόνας. Διαφορετικά χαρακτηριστικά αποτυπώνουν διαφορετικές πτυχές των δεδομένων μιας εικόνας και μπορεί ορισμένα χαρακτηριστικά να αποτυπώνουν πιο βασικές πληροφορίες απ΄ότι άλλα και επομένως να μην είναι όσο χρήσιμα για συγκεκριμένες εφαρμογές. Στα βαθιά νευρωνικά δίκτυα τα οποία αποτελούνται από πολλά επίπεδα συνελικτικών νευρώνων, τα πιο ειδικευμένα ως προς την εφαρμογή χαρακτηριστικά βρίσκονται στα πιο βαθιά συνελικτικά επίπεδα.

Τα πλέον επιτυχημένα βαθιά νευρωνικά δίκτυα είναι δίκτυα τα οποία έχουν δημιουργηθεί από μεγάλους ερευνητικούς οργανισμούς και ερευνητές με πολλές γνώσεις, ενώ έχουν εκπαιδευτεί για να αντιμετωπίζουν δύσκολα ερευντικά προβλήματα αξιοποιώντας τεράστιους όγκους δεδομένων.

Ωστόσο μπορούν να χρησιμοποιηθούν για διαφορετικές εφαρμογές από αυτές για τις οποίες δημιουργήθηκαν, απλά αξιοποιώντας τα βαθιά χαρακτηριστικά που εξάγουν από τις εικόνες ενός μικρότερου συνόλου.

Χρησιμοποιώντας αυτά τα βαθιά χαρακτηριστικά σε συνδυασμό με απλούστερες τεχνικές και αγλοριθμους, μπορούμε να πετύχουμε καλά αποτελέσματα στην κατάτμηση εικόνας με μικρότερο σύνολο δεδομένων και κατα συνέπεια μικρό χρόνο εκτέλεσης. Τεχνικές στατισκής μοντελοποιήσης, όπως Γκαουσινά Μοντέλα Μίξης (Gaussian Mixture Models) μπορούν να πετύχουν καλά αποτελέσματα προσεγγίζοντας την κατανομή των δεδομένων στα βαθιά χαρακτηριστικά.

Ταυτόχρονα, τεχνικές μείωσης διάστασης μπορούν να βοηθήσουν ώστε να συμπτυχθεί η χρήσιμη πληροφορία των χαρακτηριστικών ώστε να αποφευχθεί η *Κατάρα της Διάστασης*. Επιπλέον , απλά φίλτρα εφαρμοσμένα στο αποτέλεσμα της κατάτμησης μπορούν να βελτιώσουν το αποτελέσμα.

Σε αυτή την εργασία προτείνουμε τη χρήση βαθιών χαρακτηριστικών από νευρωνικά δίκτυα σε συνδυασμό με Γκαουσιανά Μοντέλα Μίξης και Μείωση Διάστασης για κατάτμηση εικόνας.

Λέξεις Κλειδιά: Βαθιά Χαρακτηριστικά, Γκαουσιανά Μοντέλα Μίξης, Μείωση Διάστασης.

Table of Contents

Introduction

Image segmentation is a computer vision task that requires the partitioning of an image into a number of segments, such that all the pixels of the image belong to a segment. The purpose of image segmentation is to simplify the image and to transform it into a format that is easier to analyze and extract meaningful information from. The result of image segmentation is an image, in which all pixels belong in one of the segments and all pixels in a single segment hold the same value. The pixels in each segment will typically be similar in some aspect such as color intensity or texture.

Image segmentation has many interesting applications in several fiends such as medicine, where image segmentation of medical images can be used to detect tumors and other pathologies. Image segmentation can also be used in traffic control systems for pedestrian and vehicle detection, traffic sign and traffic light detection as well as for the purposes of surveillance. It can also be used for face fingerprint and iris recognition. With the advances in computer vision techniques and the increase in processing power of recent years, image segmentation stands to be a very relevant and important computer vision task in the future.

Thresholding

The simplest form of image segmentation is thresholding on a gray scale image. If the pixel values of a gray scale image get reduced from 256 to a limited number, this will result to a segmentation of the image where pixels with similar intensity will belong in the same segment. The same can be applied to color images. If we assume that the segments of an image can be determined by the intensity of their pixels, then we could select a pixel value for which all pixels above and all pixels

below will belong in different segments; this pixel value is called *threshold*. Selecting a single threshold value will result in a segmentation of two segments, this is called *bilevel thresholding* and it can be used on text images, where there are only two major shades of gray. For more complex imagery we can select several thresholds to define several segments, this is called *multilevel thresholding*. However the pixel intensity for these thresholds can be difficult to determine due to factors such as the distribution of the shades of grey, small object and overlapping objects on the image.

Otsu's method

A highly successful thresholding method is the *Otsu's method*. This method finds a threshold value that defines two pixel classes one for the foreground and one for the background. Otsu's method does this by maximizing the variance between the classes. Variance can be thought of a measure of how disperse the pixel values are, meaning, the higher the variance, the more disperse the pixel values.

The method starts by calculating the histogram of the image. The histogram of an image is the distribution of its pixel's values and the way that we calculate it is by counting the instances of each pixel value. The histogram will have L bins which represent the unique pixel values in the image. An example of a histogram is demonstrated below:

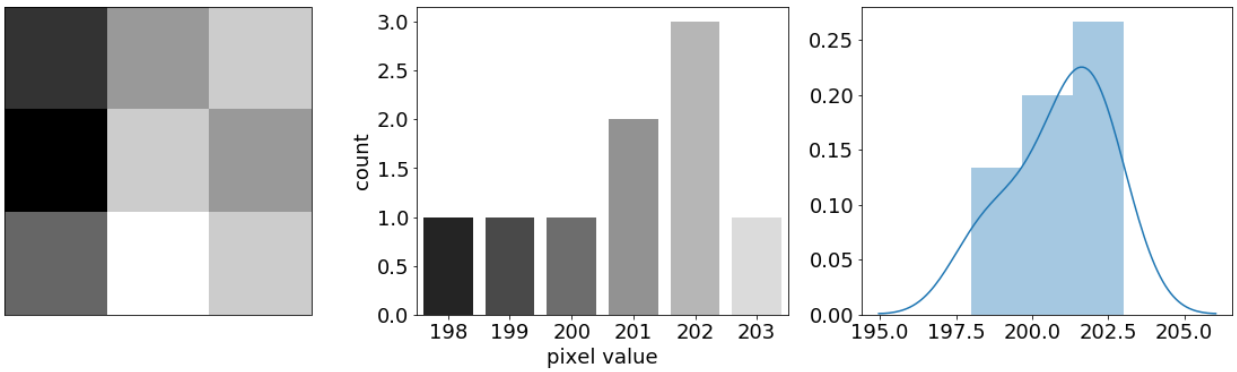


Figure 1: A 3x3 gray scale image, its histogram and its probability distribution.

Otsu's method uses the L bins of the histogram of the image to calculate the probability ω_0, ω_1 for each pixel to belong in one of the two classes. These probabilities are defined as follows:

$$\omega_0(t) = \sum_{i=0}^{t-1} p(i), \qquad \omega_1(t) = \sum_{i=t}^{L-1} p(i)$$

The method also requires the mean of each class which is defined as follows:

$$\mu_0(t) = \frac{\sum_{i=0}^{t-1} ip(i)}{\omega_0(t)}, \qquad \mu_1(t) = \frac{\sum_{i=t}^{L-1} ip(i)}{\omega_1(t)}$$

Using those quantities the method calculates the intra class variance which is defined as follows:

$$\sigma_b^2 = \omega_0(t)\omega_1(t)[\mu_0(t) - \mu_1(t)]^2$$

The method calculates the intra class variance for a specific threshold value and then repeats the calculation for all the possible threshold values in the range of pixel intensity [0,255], searching for the threshold value that minimizes the intra class variance. Minimizing the intra class variance is equal to maximizing the variance between the two classes. Otsu's method can also be used for *multilevel thresholding.*

Reddi, Rudin and Keshvan method

An extension of Otsu's method for *multilevel thresholding* can be found in Reddi, Rudin and Keshvan's method which calculates the inter class variance for several thresholds. The inter class variance is defined as follows:

$${\sigma_B}^2 = \omega_0(\mu_0 - \mu)^2 + \omega_1(\mu_1 - \mu)^2 + \cdots + \omega_n(\mu_n - \mu)^2$$

where ω_j is the probability that any pixel value belongs in class j and $\mu_j$is the mean pixel value of class j. These quantities are defined as follows:

$$\omega_0 = \sum_{i=0}^{k_1} p_i\,,\ \omega_j = \sum_{i=k_j}^{k_{j-1}} p_i,\ \ \omega_n = \sum_{i=k_i}^{255} p_i$$

$$\mu_0 = \frac{\sum_{i=0}^{k_1} ip_i}{\omega_0}, \quad \mu_j = \frac{\sum_{i=k_j}^{k_{j+1}} ip_i}{\omega_j}, \ \ \mu_n = \frac{\sum_{i=k_n}^{255} ip_i}{\omega_n}$$

The optimal values for the k thresholds are selected such that the inter class variance ${\sigma_B}^2(k_1, k_2, \dots, k_n)$ is maximized. It has been proven that thresholds that maximize the inter class variance can be calculated as follows:

$$m(0, k_1) + m(k_1, k_2) = 2k_1$$

$$m(k_1, k_2) + m(k_2, k_3) = 2k_2$$

...

$$m(k_{n-1}, k_n) + m(k_n, k_{255}) = 2k_n$$

where, $m(k_i, k_j)$ is the maximum inter class variance for classes i, j and is defined as follows:

$$m(k_i, k_j) = \frac{\sum_{x=k_i}^{k_j} xp_x}{\sum_{x=k_i}^{k_j} p_x}$$

The task of maximizing the inter class variance can be solved by finding the appropriate threshold values that will satisfy the equations above. These threshold values can be found by means of exhaustive search or by utilizing an optimization technique. The researchers propose the following technique:

First we define the number of desired thresholds n. The initial values for the n thresholds are defined as $k_i = \frac{256i}{n+1}$ for threshold i. Next, we calculate the following quantities:

$$e_1(0, k_1) = \frac{[m(0, k_1) + m(k_1, k_2)]}{2} - k_1$$

$$e_1(k_1, k_2) = \frac{[m(k_1, k_2) + m(k_2, k_3)]}{2} - k_2$$

...

$$e_1(k_{n-1}, k_n) = \frac{[m(k_{n-1}, k_n) + m(k_n, k_{255})]}{2} - k_n$$

Next, the new values for the thresholds are calculated as $k_i = k_i + [e_i]$ for $i = 1,2, \dots, n$ where $[x]$ is the nearest integer. If the maximum of $|e_1|, |e_2|, \dots, |e_n|$ is smaller than 0.5 the algorithm has converged to the final threshold values otherwise a new iteration gets executed.

Entropy based multilevel thresholding

This is a method that relies on the maximization of a criterion that is related to the entropy of an image. This way, it can be used to determine the values of the thresholds for histograms where the hills and valleys are not clear. If we want to determine k thresholds such that $T_1 \leq T_2 \leq \cdots \leq T_k$ we can calculate the entropy for every possible set of pixel values given the histogram's values p_i where$i = 0, \dots,255$. The entropy function is defined as follows:

$$E(T_1, T_2, \dots, T_k) = \ln\left(\sum_{i=1}^{T_1} p_i\right) + \ln\left(\sum_{i=T_1+1}^{T_2} p_i\right) + \dots + \ln\left(\sum_{i=T_k+1}^{255} p_i\right)$$

$$-\frac{\sum_{i=1}^{T_1} p_i \ln(p_i)}{\sum_{i=1}^{T_1} p_i} - \frac{\sum_{i=T_1+1}^{T_2} p_i \ln(p_i)}{\sum_{i=T_1+1}^{T_2} p_i} - \dots - \frac{\sum_{i=T_k+1}^{255} p_i \ln(p_i)}{\sum_{i=T_k+1}^{255} p_i}$$

The optimal threshold values are the values that maximize the entropy function $E(T_1, T_2, \dots, T_k)$. Unfortunately, this method requires exhaustive search of the threshold values, however by leveraging the values discovered in previous iterations it is possible to speed up the optimization of

the entropy function. Alternatively, the discovery of the threshold values can be achieved by utilizing the appropriate optimization technique.

Local Thresholding techniques

These techniques rely on the approximation of a threshold value for local pixel subsets of the image. More specifically, each pixel defines a neighborhood of other pixels around it. There are several parameters, such as the mean pixel value and the standard deviation, that are utilized to approximate a local threshold, which is different for each pixel. Local threshold methods are useful in cases where there is noise in the image however their drawback is that they wield different results for different parameters.

Niblack method

This method calculates a local threshold for each pixel, using the mean pixel value and the standard deviation of the pixel value of a pixel's neighborhood. The threshold for each pixel is calculated as:

$$T(x, y) = m(x, y) + k * s(x, y)$$

where $m(x, y)$ is the mean pixel value of the neighborhood defined by pixel (x, y) and $s(x, y)$ is the standard deviation of the pixel value. The k parameter is used to determine the percentage of black pixels, especially at the edges of objects. It has been found through experiments that k=-0.2 results in objects that are clearly separated from their background. The neighborhood size needs to be small enough so that local details can contribute to the threshold calculation but also large enough to reduce noise in the image. It has been found that a neighborhood of 15x15 pixels wields good results.

Bernsen method

This method calculates a local threshold for each pixel, using the mean of the minimum and maximum pixel values of a pixel's neighborhood. This calculation requires that the difference of the minimum and maximum pixel values is greater than or equal to a predetermined threshold L. This means that the local thresholding is applied on spots with sufficient contrast. If this contrast criterion is not met, this means that the pixels belong on the same class, usually the background. A more sound approach to this would be to binarize these pixels using a global threshold using Otsu's method for instance. The threshold for each pixel is calculated as:

$$T(x,y) = \begin{cases} \frac{Z_{max} + Z_{min}}{2} \; if \; Z_{max} - Z_{min} \geq L \\ Global \; Threshold \; if \; Z_{max} - Z_{min} < L \end{cases}$$

where Z_{max} and Z_{min} are the maximum and minimum pixel values respectively and L is the local threshold for a neighborhood of NxN pixels. It has been proven through experiments that N=15 and L=15 wield satisfactory results.

Sauvola's method

This method is similar with Niblack's method in that it uses the mean pixel value and the standard deviation of the pixel value to calculate the local thresholds. What's different is the method of calculation which is defined as:

$$T(x,y) = m(x,y)\left[1 + k\left(1 - \frac{s(x,y)}{R}\right)\right]$$

where k and R are usually static values with k=0.1 and R = 128.

Histogram based methods

In many real world instances, the objects in a gray scale image tend to have similar pixel values. For instance in images of text the letters all share similar pixel values, but this also happens for many other objects. This can be observed on the histogram of the image as "hills" and "valleys".

In the figure below we can see that there are two pixel values for which we have the most instances. These are the hills and they are followed by valleys.

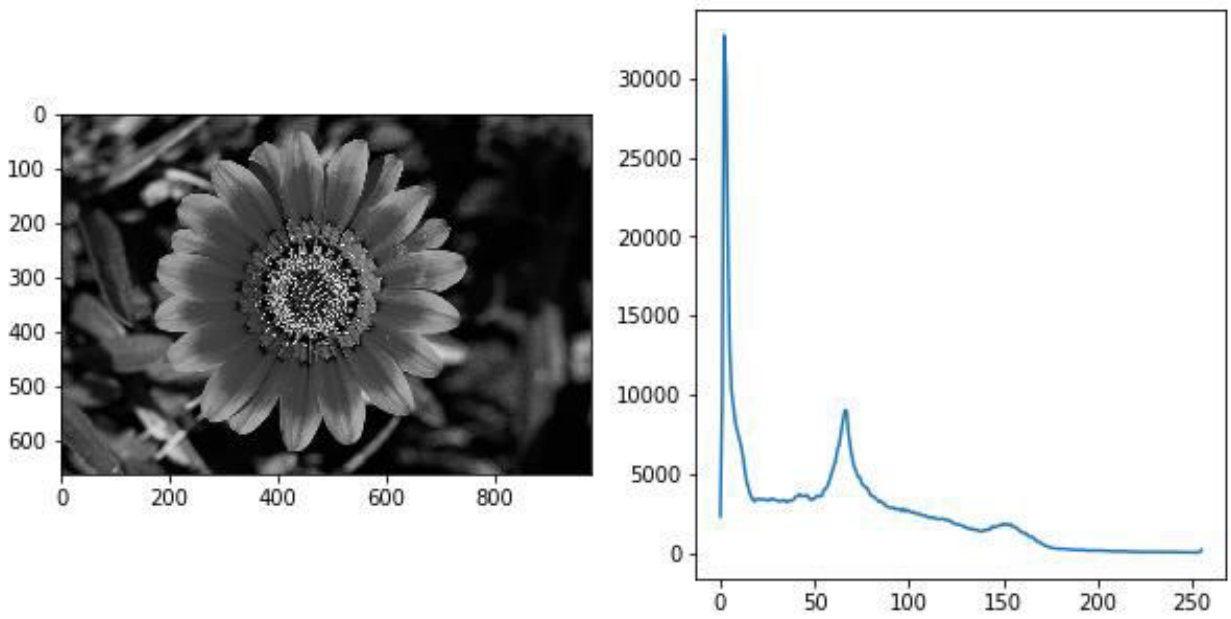


Figure 2: An image and its histogram.

It is clear that the lower pixel value hill represents the darker background and the slightly higher pixel value hill the brighter flower. However it is harder to tell what the pixel values of the valley represent.

What we can surely assume is that the closer the value of a pixel to a hill, the bigger the probability that the pixel belongs in the class that the hill represents. If we can use a method to determine the pixel value for the threshold T, then we can segment the image to two segments by applying the rule:

$$b(m,n) = \begin{cases} 1 \; if \; f(m.n) > T \\ 0 \; if \; f(m,n) \leq T \end{cases}$$

where (m,n) are the coordinates of a pixel in an image and $f(m,n)$ is the value of that pixel. This idea of exploiting the histogram's shape can also be applied in cases where there are more than two major hills.

Hill clustering

Hill clustering is a method that exploits the shape of the histogram by assuming that in a given image every major object corresponds to a hill in the histogram and obviously there are valleys between the hills. The goal of this method is to iteratively smoothen out the histogram such that the pixel values of all the hills can be approximated. It requires the maximum number of thresholds to be specified while it converges to an optimal number of thresholds within the constraint.

The hill clustering algorithm starts with the maximum number of hills for the image. The algorithm separates the histogram of the image into cells. The number of cells is calculated as N = $256/2^{t-1}$ where t is the current iteration. The frequency of each cell is calculated as $g = 0.5(g_i + g_{i-1})$ where $i = 0, \dots, N$ and $g_i$is the frequency in the previous iteration.

To determine the number of hills in each iteration the algorithm checks each pixel value in each cell and if its left and right pixel values are smaller, then it constitutes a hill and becomes that cell's value. If the left and right pixel values are equal then the middle value becomes the cell's value.

Next, if the number of cells is smaller or equal to the maximum number of hills, the algorithm has converged and the pixel values of the hills can be found by multiplying their position by2^i. Alternatively, a new iteration gets executed and the cells are further reduced as explained.

Mean Shift

Mean Shift is a non-parametric iterative classification algorithm that is commonly used on computer vision and image processing. The basic idea of this algorithm is shifting each pixel value to the mean of the pixel's neighborhood. Mean shift can be applied in a variety of applications such as classification, finding the modes of a probability density function, which is the value that appears most often in a set of data values, approximating a probability density function, image filtering, color reduction and image segmentation as well as object tracking.

The fundamental principal of the mean shift algorithm is the consecutive shifting of a data point towards the mean value of the data points within the neighborhood of the data point. From a mathematical point of view, the mean shift algorithm seeks the modes of a probability density function. Assuming a dataset size S and n the number of d-dimensional vectors $x_1, x_2, \dots, x_n$, we define a kernel function K($\mathbf{x}$) which represents the contribution of vector $\mathbf{x}$ to the local approximation of the mean value. The local mean value at the position of x vector with that given kernel function is defined as follows:

$$m(\boldsymbol{x}) = \frac{\sum_{i=1}^{n} K(\boldsymbol{x} - \boldsymbol{x}_i)\boldsymbol{x}_i}{\sum_{i=1}^{n} K(\boldsymbol{x} - \boldsymbol{x}_i)}$$

The $m(\boldsymbol{x}) - \boldsymbol{x}$ portion is called mean shift and the mean-shift vector of vector $\mathbf{x}$ is defined as $M(\boldsymbol{x}) = m(\boldsymbol{x}) - \boldsymbol{x}$. The mean shift algorithm therefore constantly shifts locally the data point towards the mean value of the distribution. In other words, in each iteration $\boldsymbol{x}$ is equated to $m(\boldsymbol{x})$

and the algorithm terminates when $\boldsymbol{x} = m(\boldsymbol{x})$. The sequence of shifts of a data point defines a *trajectory* and when the above process is applied to many points in a dataset then we have a simultaneous shift towards the mean values of the distribution on a local level.

Typically the kernel function $K(\boldsymbol{x})$ is a function of $\|\boldsymbol{x}^2\|$, in other words $K(\boldsymbol{x})$ can be defined as:

$$K(\boldsymbol{x}) = \mathrm{g}(\|\boldsymbol{x}^2\|)$$

where the function g is called the *profile* of $K(\boldsymbol{x})$ and it needs to satisfy the following properties:

1. It always holds a non-negative value.
2. It is a non-ascending function or $g(a) \geq g(b) if\ a < b$.
3. It is partially continuous and $\int_0^\infty g(x)dx < \infty$.

For a dataset of n points $\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_n$ within a R^d Euclidean space of dimension d, we can define the $\hat{f}_k(\boldsymbol{x})$ function which is known as the kernel density estimator and is defined as follows:

$$\hat{f}_k(\boldsymbol{x}) = \frac{1}{nh^d}\sum_{i=1}^{n} K\left(\frac{\boldsymbol{x} - \boldsymbol{x}_i}{h}\right)$$

If we replace the kernel function form the equation above then $\hat{f}_k(\boldsymbol{x})$ becomes:

$$\hat{f}_k(\boldsymbol{x}) = \frac{1}{nh^d}\sum_{i=1}^{n} g\left(\left\|\frac{\boldsymbol{x} - \boldsymbol{x}_i}{h}\right\|^2\right)$$

The h parameter is crucial as it determines the size of the kernel K and its value is definitive for the final result of mean shift. From the equation above we can calculate the slope of $\hat{f}_k(\boldsymbol{x})$ as follows:

$$\widehat{\nabla}\hat{f}_k(\boldsymbol{x}) = \frac{2}{n}\frac{1}{h^{d+2}}\left[\sum_{i=1}^{n} g\left(\left\|\frac{\boldsymbol{x} - \boldsymbol{x}_i}{h}\right\|^2\right)\right]\left[\frac{\sum_{i=1}^{n} g\left(\left\|\frac{\boldsymbol{x} - \boldsymbol{x}_i}{h}\right\|^2\right)\boldsymbol{x}_i}{\sum_{i=1}^{n} g\left(\left\|\frac{\boldsymbol{x} - \boldsymbol{x}_i}{h}\right\|^2\right)} - x\right]$$

We can observe that

$$m(\boldsymbol{x}) = \frac{\sum_{i=1}^{n} g\left(\left\|\frac{\boldsymbol{x} - \boldsymbol{x}_i}{h}\right\|^2\right)\boldsymbol{x}_i}{\sum_{i=1}^{n} g\left(\left\|\frac{\boldsymbol{x} - \boldsymbol{x}_i}{h}\right\|^2\right)}$$

This means that we can express the mean shift vector as follows:

$$M(\boldsymbol{x}) = \frac{1}{2}h^2 c\frac{\widehat{\nabla}\hat{f}_k(\boldsymbol{x})}{\hat{f}_k(\boldsymbol{x})}$$

where c is a normalized constant. From the equation above we can deduce that the mean shift vector is analogous to the slope of the kernel density estimator $\widehat{\nabla}\hat{f}_k(\boldsymbol{x})$. Using this insight it can be

proved that the mean shift process always results in a local maximum meaning a point with a slope of zero. In other words, the mean shift vector $M(\boldsymbol{x})$ always has the direction of the maximum increase in the density of data point within the space.

The quality of the kernel function can be evaluated using the *Mean Square Error* between the real distribution and $K(\boldsymbol{x})$. It has been proven that the Mean Square Error is minimized when we use the Epanechnikov kernel function which is defined as follows:

$$K(x) = \begin{cases} \frac{1}{2C_d}(h+2)(1-\|\boldsymbol{x}\|^2)\ if\ \|\boldsymbol{x}\| \leq 1 \\ 0\ otherwise \end{cases}$$

Some other popular kernel functions are the uniform kernel, defined as follows:

$$K(\boldsymbol{x}) = \begin{cases} C_d\ if\ \|\boldsymbol{x}\| \leq 1 \\ 0\ otherwise \end{cases}$$

And the Gaussian kernel, defined as follows:

$$K(\boldsymbol{x}) = \frac{1}{\sqrt{(2\pi)^d}} e^{-\frac{1}{2}\|\boldsymbol{x}\|^2}$$

The mean shift algorithm can be used for the purposes of color reduction on images, either standalone or in combination with other color reduction techniques in which case it used for the initial reduction of colors on an image with the prospect of further reduction using other techniques. There are two main issues that need to be addressed and these are the size of the kernel that mean shift will use and the value of the h parameter. Usually we choose the Gaussian kernel in which case the mean shift vector is calculated as:

$$M(\boldsymbol{x}) = \frac{\sum_{i=1}^{n} e^{-\frac{1}{2}\left\|\frac{\boldsymbol{x}-\boldsymbol{x}_i}{h}\right\|^2} \boldsymbol{x}_i}{\sum_{i=1}^{n} e^{-\frac{1}{2}\left\|\frac{\boldsymbol{x}-\boldsymbol{x}_i}{h}\right\|^2}} - \boldsymbol{x}$$

If we have m features and we select a different value for h for every feature, then in each iteration the point y_{k+1} will be determined by the value of the previous point y_k according to the following equation:

$$y_{k+1} = \frac{\sum_{i=1}^{n} x_i exp\left(-\frac{1}{2}\sum_{m=1}^{d}\left(\frac{y_k^m - x_i^m}{h_m}\right)^2\right)}{\sum_{i=1}^{n} exp\left(-\frac{1}{2}\sum_{m=1}^{d}\left(\frac{y_k^m - x_i^m}{h_m}\right)^2\right)}$$

For simplicity, the uniform kernel is also commonly used, which can be matched to an orthogonal area or mask on the image, on account of image quantization. In addition we need to select the color space which will use. For this, it is better to use a homogenous color space such as La^*b^*. One

additional parameter to this is the color distance which will be used, which for in each iteration determines which pixel's color, will be considered for the new center of the class.

Image segmentation can be applied using the following algorithm:

First, project the image into a d-dimensional space such that each pixel $x_i \in R^d$. Initialize j=1 and $y_{i,j} = x_i$. Then calculate $y_{i,j+1}$ as we described above and iterate until the algorithm converges and then assign $z_i = y_{i,c}$, where $y_{i,c}$, the value of convergence. Next, initialize the classes $\{C_p\}, p = 1, \dots, m$ with $m = 1,2, \dots, d$, that are defined by uniting all the z_i that are closest to h_m out of the m features. For every $i = 1,2, \dots, n$ assign $L_i = \{p | z_i \in C_p\}$. Finally eliminate the local areas that contain smaller number of pixels than a predetermined threshold.

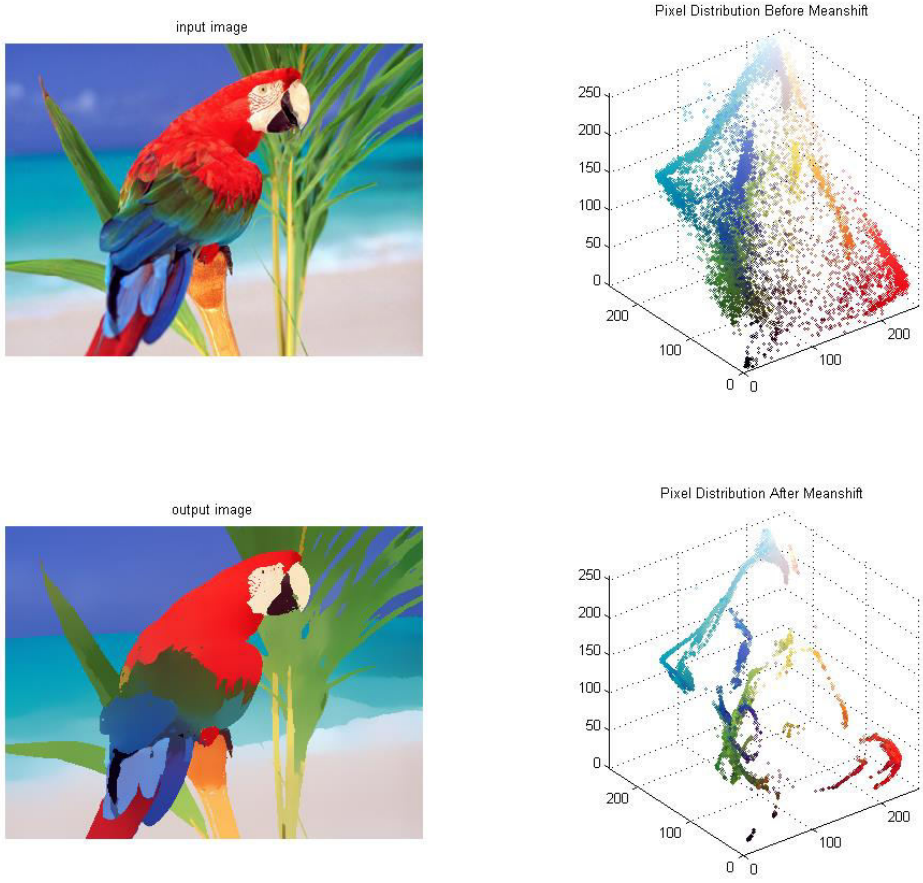


Figure 3: Results of image segmentation using mean shift.

Segmentation using projections

Projecting images into various directions can give us valuable information about the spacial distribution of the objects they depicts. Segmenting an object in an image can be achieved using a projection such as horizontal or orthogonal projection or even a projection of a different angle. The application of this method on gray scale images requires the binarization of the image or the use of a projection that is related to a specific feature and is able to separate the objects from the background of the image. This technique however doesn't work very well when there are multiple objects in an image or if the objects are not well discernable from the background.

Region Growing

Region growing is a useful technique if we have some sort of initial segmentation such as grouping up neighboring pixels with similar pixel values. In these cases we can perform region growing by merging them with their similar neighboring regions. The similarity between the regions can be defined in a variety of ways. For instance similarity can be defined by comparing the means of the pixel values and the standard deviations or by some criteria related to the geometric features of the regions as well as other features of the regions.

Image segmentation using region growing can be performed by separating an image into different segments which will be represented by different regions.

Initially the image is divided into separate regions which have consistent shades of gray. These regions start off as single pixels which are called seed pixels. Similar neighboring regions are united as one region and this keeps happening iteratively until the regions do not meet the similarity criteria. The geometric criteria for the region growing are defined as follows:

$$\frac{D}{\min(P_i, P_j)} > \xi$$

where P_i and P_j are the perimeters of the regions R_i and R_j respectively. D is the number of weak boundary locations, which is defined as the pixels in each side of the regions R_i and R_j whose absolute difference is smaller than a predetermined threshold. ξ is a constant which typically takes a value of 0.5.

The union of two regions is permitted as long as they meet the following criterion:

$$\frac{\mathrm{D}}{C} > \delta$$

where C is the length of the border between the two regions and δ is a constant which typically takes the value of 0.7. According to this criterion, two regions are united when their borders are weak enough. Very often this criterion is combined with the first criterion to reduce the number of regions that need to be examined.

The most important factor in region growing is the selection of the optimal criterion for union or not of two regions. A variation of region growing starts with a large number of regions and iteratively unites neighboring regions using a criterion for homogeneity and border length. That way, small regions with similar pixel values (or alternatively similar mean pixel values) and small border length, are united to form larger regions. As a termination criterion, this variation uses a maximum number of regions which needs to be specified.

Image Segmentation using Division and Union

This method divides the image into four quadrants and then examines whether or not there is homogeneity in each of the quadrants. If there is no homogeneity in a quadrant, then that quadrant is divided into another four quadrants. If two quadrants are homogenous then they united. This process is repeated until the final segmentation result.

Template Matching method

This method performs image segmentation on an image by comparing it to an existing template. When the object for detection is specified, we can separate the image into two classes: one that contains the object and one that doesn't. Given an image I(x, y) with MxN dimension and T(x, y) a template image with KxL dimensions smaller than MxN, we can assume that the template image defines a rectangle window W, such that $(K, L) \in W$. Let's assume that the template image T(x, y) is convoluted with the original I(x, y). In each position (x, y), we can calculate the Euclidian distance as follows:

$$E^2(x,y) = \sum_i \sum_{\substack{j \\ (i,j)\in W}} [I(x+i, y+j) - T(i,j)]^2$$

where (x, y) corresponds to the upper left side of the template image T(x, y). The equation above can be rewritten as follows:

$$E^2(x,y) = \sum_i \sum_{\substack{j \\ (i,j)\in W}} [I(x+i, y+j) - 2I(x+i, y+j)T(i,j) + T^2(i,j)]^{-2}$$

We can observe that the term $\sum_i \sum_j T^2(i,j)$ is constant. In addition if the term $\sum_i \sum_j I^2(x+i, y+j)$ is approximately constant on the image, then the following term:

$$c(x,y) = \sum_i \sum_{\substack{j \\ (i,j)\in W}} I(x+i, y+j)T(i,j)$$

is a measure of homogeneity between the template image and the part of the original image inside the window W. Consequently, in order to minimize the Euclidean distance we need to find the position that maximizes $c(x,y)$. In other words, the optimal position is the position for which the following is true:

$$max \sum_i \sum_{\substack{j \\ (i,j)\in W}} I(x+i, y+j)T(i,j)$$

The $c(x,y)$ function effectively represents the *cross-correlation* of the I and T images. The use of *cross-correlation* however has a few drawbacks. If the pixel values on the image shift drastically from position to position then the term $\sum_i \sum_j T^2(i,j)$ is not constant. Another drawback is that the size of the $c(x,y)$ function depends on the size of the template image T. For this reason, it is better to use the Euclidian distance for finding the optimal position. Alternatively we can use a normalized version of *cross-correlation*. For instance if the term $\sum_i \sum_j T^2(i,j)$ is constant, the minimization of the Euclidean distance is defined as:

$$min \left[1 - 2 \frac{\sum_i \sum_{\substack{j \\ (i,j)\in W}} I(x+i, y+j)T(i,j)}{\sum_i \sum_{\substack{j \\ (i,j)\in W}} I^2(x+i, y+j)} \right]$$

In this equation the first term is constant, so the optimal position can be defined as follows:

$$minC = \left[\frac{\sum_i \sum_{\substack{j \\ (i,j)\in W}} I(x+i, y+j)T(i,j)}{\sum_i \sum_{\substack{j \\ (i,j)\in W}} I^2(x+i, y+j)} \right]$$

A good idea is to normalize the pixel values of the image using the mean pixel values in each position. In that case the above equation can be rewritten as follows:

$$minC = \left[\frac{\sum_i \sum_{\substack{j \\ (i,j)\in W}} [I(x+i, y+j) - \bar{I}(i,j)][T(i,j) - \bar{T}]}{\sum_i \sum_{\substack{j \\ (i,j)\in W}} [I(x+i, y+j) - \bar{I}(i,j)]^2} \right]$$

This application of the template matching method can be useful for images with many objects and images with text and graphics.

Flooding Technique

This segmentation technique relies on the flooding process known as Watershed Segmentation and it exploits the 3Dimennsional terrain that corresponds to each gray scale image. The main purpose of the flooding algorithm is finding the flooding lines, also known as watersheds, which can be used to separate the different regions of an image, thus achieving image segmentation. We need to point out that many times the regions of an image can present small deviations in the shade of gray, in other words they have small gradient values. For that reason, it is common to see the flooding technique performed not on the original image but the gradient image instead.

The main idea of the flooding technique relies on the assumption of on image on three dimensions, in other words there is (x, y) which express the width and height of a pixel and there is also z which expresses the pixel value on that position. Usually the flooding technique is applied on the gradient image because the deviations in the shade of gray have a small gradient. In fact it is common to use a Gaussian filter to denoise the initial image and apply the flooding technique on the gradient of the filtered image. There are three types of points that are considered important:

a) The points that belong in a local minimum.
b) The points close to a specific local minimum.
c) The points close to multiple local minima.

Any point that satisfies the b) criterion is called *catchment basin* or watershed of the specific local minimum. Any point that satisfies the c) criterion are the points that form what is called *watershed line*, which is the line that separates the regions.

In geography, a watershed is any area of land where all the water on that area flows to a specific spot. Thus a catchment basin can be an area that allows water to flow towards a river or a lake or the sea.

Assuming a gray scale image in its 3 dimensional form, the algorithm starts by assuming that we are dropping water on the topological diagram of the image, from lower height areas all the way up to the highest area. The water level will rise homogenously because of gravity and during the rise of the water level we can locate the high areas which separate the catchment basins. These areas constitute what are known as *dams*. The flood will eventually reach a point where only the high

areas of the dams will be visible on the surface of the water. The tips of the dams form continuous lines which separate the regions thus achieving segmentation of the image.

Texture Image Segmentation

In some cases, the background or even the object depicted on an image has a certain texture. In those cases we also observe high density of edges and region based segmentation techniques are not effective. A good idea then, is to use techniques that rely on detecting texture characteristics such as statistical morphological or plasmatic features. Using these features as well as the appropriate classifier, we can achieve segmentation on images with high texture.

Active Contours (Snakes)

This technique can provide good segmentation that is not too large or too small and has the correct shape in cases of texture heavy images by utilizing curves or contours which represent the boundary lines.

First, these contours are initialized, either automatically, manually or semi automatically. The initialization of the contour is a crucial factor for the quality of the segmentation and poor initialization can skew the results, especially when there are small neighboring objects or in cases when the image is noisy. It is best for the curves to be initialized on a "cleaner" part of the image if possible. Some snake algorithms require the initialization of the curves entirely inside or entirely outside of the image. The initialization can be improved by involving a human, who can draw a simply curve such as an eclipse on the object or by roughly marking some boundary points.

In order for the contour to act as the boundary line of the segments, it will need to change its position on the image. There are two common philosophies about how to move the contour, Energy minimization and Partial differential equations.

Energy minimization requires an ad-hoc energy function that describes how well the curve matches the image. For the energy function, visual image boundaries represent a low energy state for the curve. Provided the energy function is defined, all we need to do is minimize the function using some numerical optimization method, which will usually wield a local minimum. This is another reason why a good initialization is important, because if the starting state is relatively low energy then the optimization will wield better results. The curve is usually represented as a set of sequentially connected points and each point is connected to its two neighboring points while it is usually closed, meaning that the first and last points are connected. It is important to point out that each point of the curve is represented by a pixel but the points change constantly as the algorithm

goes through the iterations. On top of that neighboring points are almost always separated by multiple pixels.

For each iteration the value of the energy function changes as the curve moves around the image. This value is called the snake's energy. It can be defined by two terms, the external energy and the internal energy such that $E = E_{external} + E_{intenral}$.

The external energy, also known as image energy, is designed to capture the desired image features. An example of external energy is defined by the following equation:

$$E_{external} = \sum_{i} exp(-\|\nabla I(\boldsymbol{X}_i)\|)$$

where $\boldsymbol{X}_i$ is the location of point i and $I(\boldsymbol{X}_i)$ is the value of that point on the image. This equation measures the gradient magnitude at the location of each point.

The internal energy, also known as shape energy, is designed to reduce the curvature of the curve and the prevent outlier points from manifesting. An example of external energy is defined by the following equation:

$$E_{intenral} = \sum_{i,j} a\|\boldsymbol{X}_i - \boldsymbol{X}_j\| + \beta\|\boldsymbol{X}_{i-1} - 2\boldsymbol{X}_i + \boldsymbol{X}_{i+1}\|$$

where i, j are two neighboring points of the curve and a, β are the constants, usually with $a < \beta$. This equation minimizes the distance between the points and the curvature of the curve.

When moving the curve, there are usually two scenarios, each point can move in any direction or the curve can shrink. Moving each point is useful when the curve is close to the correct boundary while shrinking the snake is useful when the curve has been initialized with a large number of points. Other scenarios are also possible. When minimizing the energy function, there are several good choices for the optimization technique such as dynamic programming or simulated annealing. If we select one of these methods we will require a finite number of possible states, typically sampled, which means that if the best solution is not included within the sample set, then we won't find it.

The alternative method for moving the curve on the image is Partial differential equations. This method starts the curve expanding or contracting while the points on the curve move more slowly as they the curvature increases as well as when they lie on image edginess. The curve ideally stops moving when it lies on the appropriate image boundaries. Partial differential equations operate on discrete time steps and the points move normally to the curve on each time step. The curve points move a certain distance that is determined by their speed which is defined by a speed function. The speed function is usually a product of internal and external terms such that
$s(x, y) = s_{intenrnal} * s_{external}$.

The internal speed or shape speed is typically defined as follows:

$$s_{intenral}(x, y) = 1 - \|\kappa(x, y)\|$$

where $\kappa(x, y)$ measures the curvature at the point (x, y).

The external speed or image speed is typically defined as follows:

$$s_{external}(x, y) = (1 + \Delta(x, y))^{-1}$$

where $\Delta(x, y)$ measure the image's edginess at the point (x, y).

The limitation of Partial differential equations is that curvature measurements are usually sensitive to noise due to the fact they use second derivatives. On top of that, the curve representations don't allow an object to split, which can be a problem when tracking an object through multiple time frames. Another problem is that the curve might cross itself. To prevent such occurrences from happening we only allow the curve to grow.

Clustering methods

Image segmentation can be achieved by performing clustering on an image's tokens such as pixel values or texture elements or surfaces. By clustering the image tokens that are similar to each other, according to some specified definition of similarity, we can separate the image into segments that represent specific attributes of the image. Image segmentation via clustering is a powerful technique for unlabeled data, in other words, cases where we don't know what kind of objects an image depicts. Clustering can help us discover patterns such as structures and groupings that would be hard to obtain with alternative techniques.

K Means clustering

The simplest and most popular clustering algorithm, K Means, relies on the minimization of the Mean Squared Distance of all the elements in each cluster from the center or *centroid* of that cluster. The process can be expressed mathematically as follows:

$$\min\left(\sum_{x \in S_j(t)} |x - z_j|^2\right)$$

where $S_j(t)$ is the cluster on the t-th iteration, z_j is the centroid of that cluster and $|°|$ is the Euclidian distance. Therefore the function that needs to be optimized for all K clusters is the defined as follows:

$$J(Z) = \sum_{j} \sum_{x \in S_j(t)} |x - z_j|^2$$

By calculating the partial derivative of $J(Z)$ we have:

$$\frac{\partial J(Z)}{\partial z_j} = \sum_{x \in S_j(t)} (x - z_j) = 0 \; \forall j$$

Consequently, the optimized function for the centroids of the clusters becomes as follows:

$$z_j = \frac{1}{n_j} \sum_{x \in S_j(t)} x$$

where n_j is the number of elements in the cluster S_j on the t-th iteration. In other words, z_j is the mean value of the elements of each cluster. The K Means clustering usually involves the following steps:

First we need to determine the number of clusters K. Next we randomly or by approximation select K points from the data, which will be the initial centroids of the K clusters. For all the rest of the points in the data we calculate the Euclidian distance between them and the centroids and we assign each point to the cluster represented by the centroid with the minimum distance. We update the value of the centroids as the mean of all the points in each cluster. We repeat this process until the algorithm has converged.

The optimization technique will in most cases result in a local minimum for the Means Square Error, which means that the optimization might never find a global minimum. Thus a convergence criterion needs to be enforced. Usually the convergence criterion is a certain maximum number of iterations or a threshold for the change in the Mean Square Error.

Figure 4: K Means clustering for different numbers of clusters.

One of the limitations of K Means is that the initialization of the centroids greatly affects the result of the clustering. For that reason, it is common to perform the algorithm on several different initializations of the centroids. The C Means algorithm can improve on that limitation.

C Means clustering

This algorithm is a variation of the K Means algorithm. The difference is that initially we assign all the points of the data into the K clusters. This is done by some approximation or in cases where such an approximation is impossible, randomly. The rest of the algorithm is identical to K Means. This variation is useful in cases where there is a rough approximation that allows us to initialize the data points in more favorable positions.

Soft clustering

The main difference between soft or fuzzy and hard clustering techniques is the fact that each element can belong in multiple clusters. More specifically, assuming M data points $x_1, x_2, \dots, x_M$ that belong in a set S, we search for the clusters $S_1, S_2, \dots, S_K$ such that every data point is clustered in a way that allows $S_1 \cup S_2 \cup \dots \cup S_K = S$ without however having $S_i \cap S_j = \emptyset, \forall i \neq j$.

Essentially, instead of assigning a data point to a specific cluster like how the K Means algorithm does, soft clustering gives each data point a probability that it belongs to every cluster of the clustering model.

The benefit of soft clustering compared to hard clustering is the ability to cluster overlapping data or data with some shared features. This gives a soft clustering technique the ability to model complex data, such as real world images as well as features extracted from images.

Fuzzy C Means

As the name suggests, this algorithm is a fuzzy variation on the C Means algorithm. The error function it minimizes is defined as follows:

$$J(U, V) = \sum_{i=1}^{c} \sum_{k=1}^{n} u_{ik}^{m} |x_k - v_i|^2$$

where $x_1, x_2, \dots, x_n$ are the n data points, $V = \{v_1, v_2, \dots, v_c\}$ are the cluster centers, m is a predetermined parameter that is known as *fuzzifier* and $U = [u_{ik}]$ is a $c \times n$ matrix known as the

contribution matrix where u_{ik} represents the degree to which element k of the n input data points belongs to the cluster i. Each u_{ik} needs to satisfy the following requirements:

$$0 < u_{ik} \leq 1, i = 1, \dots, c, k = 1, \dots, n$$

$$\sum_{i=1}^{c} u_{ik} = 1, k = 1, \dots, n$$

$$0 < \sum_{k=1}^{n} u_{ik} < n, i = 1, \dots, c$$

In order to cluster some data points using the Fuzzy C Means algorithm we need to go through the following steps:

First we initialize the number of clusters c such that $2 \leq c \leq N$, the maximum number of iterations T, the fuzzifier's value $m > 1$ and the threshold for convergence $\varepsilon > 0$. Next we initialize the contribution matrix $U^{(0)}$ either randomly or by some approximation. For each iteration, we calculate the centers of the clusters as follows:

$$v_i = \frac{\sum_{k=1}^{n} u_{ik}^m x_{ik}}{\sum_{k=1}^{n} u_{ik}^m}, i = 1, \dots, c$$

which is essentially the mean of x_{ik}, multiplied by u_{ik}^m. Then we calculate contribution matrix $U^{(t)}$ as follows:

$$u_{ik} = \frac{\left[\frac{1}{|x_k - v_i|^2}\right]^{1/(m-1)}}{\sum_{j=1}^{c}\left[\frac{1}{|x_k - v_j|^2}\right]^{1/(m-1)}}, i = 1, \dots, c, k = 1, \dots, n$$

We stop the process when the following criterion is met:

$$A_i = \sqrt[p]{\det(S_i)} S_i^{-1}$$

If the criterion is not met, we execute the next iteration.

Maxmin clustering

This clustering algorithm is a very useful one because it allows us to estimate the number of clusters. It's automatic and unsupervised process of cluster estimation makes it ideal for initializing the clusters for other clustering algorithms such as K Means. Given N features and K samples such that $f_k = \left[f_{k,1}, f_{k,2}, \dots, f_{k,N}\right]^T$ with $= 1,2, \dots, K$, the algorithm executes the following steps:

First, we assign the first vector f_1 as the center z_1 of the first cluster. As center z_2 of the second cluster we assign the most far away from z_1 sample. For the rest of the samples, we calculate their distances from the centers of the other clusters. The smaller of these distances is the known as maxmin distance, D_{maxmin}. If $D_{maxmin} \ll \|z_1 - z_2\|$ then the number of clusters is two. If D_{maxmin} is comparable with the distance between the centers z_1 and z_2 then we initialize a new cluster with center z_3, the most far away sample. We repeat the process until convergence.

ISODATA clustering

ISODATA is the acronym for *Iterative Self Organizing Data Analysis Techniques*. This clustering algorithm requires several parameters to be predetermined. These parameters are the maximum number of clusters M, the minimum number of elements in each cluster, the maximum standard deviation σ_x, the minimum distance between the clusters δ, the maximum number of clusters that can be merged L and the maximum number of iterations I.

This clustering algorithm is considered to be an optimized version of K Means and C Means algorithms. Similarly to these methods the criterion on which clustering will be performed is minimizing the Mean Square Error across all the clusters. However there are some variations on the aforementioned algorithms, for instance, the number of clusters is not predetermined, only the maximum number of clusters is. In addition, some clusters with too few elements may be rejected since the minimum number of elements for each cluster η is predetermined. The maximum standard deviation is used to ensure that similar elements are close to each other, in what is known as *reliability property*. Finally the minimum distance between clusters δ is used to ensure that dissimilar elements are far apart to each other.

This clustering algorithm executes the following steps:

First we initialize the maximum number of clusters and their corresponding centers either randomly or by some approximation. For each element in the data, we calculate the Euclidean distance with each of the cluster centers and we assign that element to the closest center's cluster. We check if any of the clusters has fewer than η elements and if it does we break that cluster apart. We calculate the standard deviation for each cluster and if it higher than σ_x then we separate the cluster into two groups, S_1 and S_2 where S_1 contains all the elements with higher mean value than the mean value of the cluster and S_2 contains other elements with lower or equal mean value than that of the cluster. If the distance between S_1 and S_2 is higher than δ then S_1 and S_2 constitute

separate clusters. Next we calculate the distances between the clusters and if any of them are smaller than δ then we merge these clusters, as long as the number of merges does not exceed L, and calculate the new center. This process will repeat for I iterations or until there is not a significant enough change in the Mean Square Error.

Neural Networks

Neural networks have become a significant field of research over the past years. They are used in a wide range of applications, mostly for pattern recognition, and this can serve the purposes of image segmentation, greatly improving the results over traditional techniques, like those we have mentioned. Artificial Neural Networks, especially deep learning techniques have seen a rise in popularity over the last decade due to several factors. These factors are the increase in computing power of CPUs and GPUs alike, the increase in the available data, also known as *Big Data* and the fact that neural networks are able to exploit parallel computing, some researchers have even used the term *Embarrassingly Parallel*.

Neural networks are inspired by biological systems, however as research makes progress the connection starts to fade. Neural networks are based on a collection of connected units or nodes that are called neurons. Each connection, like the synapses in a biological brain can transmit a signal to other neurons. A neuron that receives a signal processes it and then transmits it to other neurons it is connected to. The signal of each connection is usually a real number of a tensor of real numbers. As for the process of each neuron, it is usually a nonlinear function that is applied on the signal input. Each connection between neurons has a *weight* value which is like a multiplier to the output of a neuron. The purpose of these weights is to regulate the effect of each neuron onto the next neuron it is connected to. In other words, the weight of each connection increases or decreases the strength of the signal that a neuron outputs. Sometimes neurons have a threshold such that if the aggregate of the signal does not meet the threshold, it is not transmitted to the next neuron. Neurons are grouped up into layers where each layer's neurons may perform a different function on their inputs. The signals from the neurons travel from lower layer neurons to higher lever neurons until they reach the last layer, which gives the final output of the network.

Neural networks initially have very poor performance in the task that they are built for and so they need to be trained on many examples or data points that are relevant to their task. This is one the reasons why neural networks have seen such a rise in popularity in recent years. It is because the increase in data quantity that is produced on the internet nowadays, has provided researchers with a vast quantity of data on which neural networks can be trained on and achieve incredible results.

Consequently, the weight values of all the connections between neurons in the network, the processes that each neuron performs as well as the training process essentially determine the functionality of a neural network.

Neural networks are defined by the topology of the network, the attributes of the neurons, in other words what kind of process they implement and finally the training rules and methods. Some of the more exciting tasks that neural networks have seen breakthroughs in include classification, clustering optimization of systems with many variables, function approximation, regression, time series analysis and associative memories.

The most successful applications of neural networks in the context of computer vision are image segmentation, image classification and clustering.

The two main categories of neural networks in terms of topology is Feed-forward networks which do not contain any loops and Recurrent or feedback networks which contain loops or recursions. Different topologies result in different behavior of the neural network.

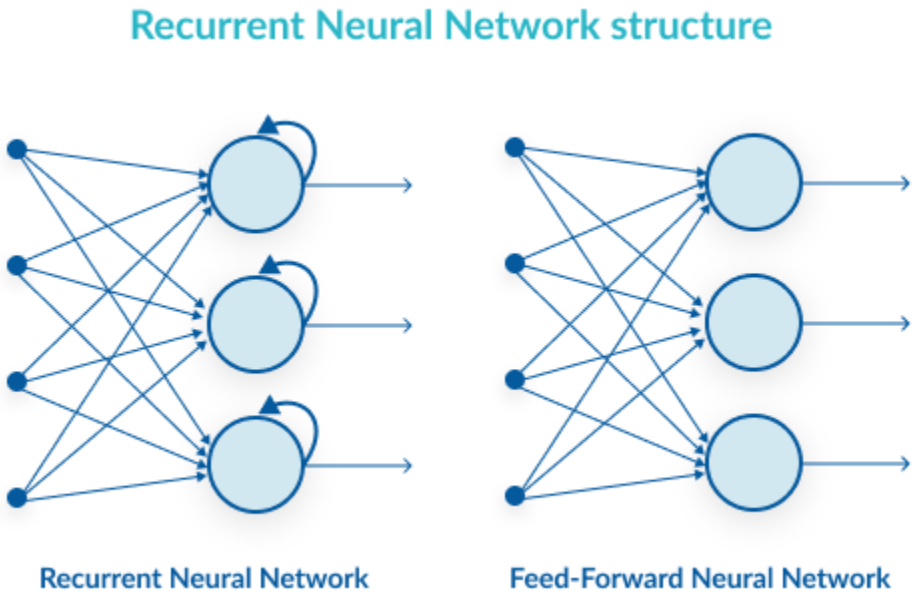


Figure 6: Feed-Forward neural network architecture and Recurrent neural network architecture.

In the most common type of neural network, feed-forward networks, we usually observe multiple layers of neurons which are most of the time fully connected. Feed-forward neural networks are considered static in the sense that they come up with a set of outputs, one for each input, as opposed to a sequence of outputs, one for each input. In addition, Feed-forward networks don't possess any sort of memory, as the response of the network to a specific input is not affected by its previous state. Recurrent neural networks however can be thought of as dynamic systems because when a new data instance is input into the network, it produces a response which loops back into the network, affecting the future inputs of the neurons and leading the network into a new state.

A fundamental attribute of intelligence is the ability to learn and neural networks are designed with this attribute in mind. Learning is a term too broad to be able to completely define but in the context of neural networks, learning is the task of readjusting or calibrating the weights of the connection between the neurons of the network and in some altering the architecture of the network itself. This process is called *training* of the neural network and the goal is to improve the neural network's performance in the task it was designed to solve. A neural network is usually able to adjust the weights of its connections based on the feedback it receives from the data points it uses for training. The performance of a neural network gradually increases as the weights are adjusted iteration by iteration. The ability of neural networks to learn automatically from the data points that they use for training makes them quite potent for solving pattern recognition tasks. Neural networks improve on traditional techniques by not requiring the human researchers to explicitly determine the rules of a solution or the exact parameters, but instead "figuring out" the rules and parameters automatically, which allows for the invention of solutions that humans are simply unable to derive with conventional means. A neural network is able to learn the underlying patterns of the data from the correlations between the inputs and the responses that the network gives. There are three main learning methods: Supervised learning, unsupervised learning and Reinforcement learning as well as some hybrid methods or methods that in some way combine techniques from the other learning methods.

In supervised learning the neural network is provided with a single feedback signal that is related to the performance of the network on the current task. It is important to point out that the human supervision does not involve the rules and adjustments that will be made to the weights of the network but only on the response of the network compared to the desired response. This naturally requires the training data to come in the form of pairs of input and desired output. An example of this that is related to computer vision would be optical recognition of digits, where a neural network receives an image as input and has to give the digit that the image depicts as the response.

In such a system the supervision that a neural network may require is the ground truth of which digit is depicted in the input image such that the network can compare its response and the correct response and adjust its weights accordingly. This process can be conceptualized as a teacher that only tells the network how wrong it was in its response.

Unsupervised learning is a learning method that requires no feedback given to the neural network regarding how correct or how accurate its response is for each data point. Instead of responding to feedback, unsupervised learning searches for commonalities and reacts based on the existence or absence of such commonalities. This learning method in essence detects patterns in the data by searching the underlying structures of the data and the correlations between them.

A hybrid learning method may combine supervised and unsupervised learning. Some of the weights are determined by the supervised learning portion and the rest by the unsupervised learning. In the case of neural network clustering methods an important issue that needs to be addressed is the ideal number of clusters. This issue can be circumvented using a Self Organizing Map neural network.

The Perceptron

This neural network is a supervised learning network which can be used for separating non linearly separable classes. A single perceptron is one of the first attempts towards the creation of intelligent learning machines that are based on simple units. It was initially proposed in 1943 from researchers McCulloch and Pitts as a computational prototype for an artificial brain neuron. Later Rosenblatt designed the perceptron neural network as a mechanism that can express the pattern recognition capabilities of biological vision. According to McCulloh and Pitts' model the response of a neuron is binary. More specifically a neuron receives the input values $x_1, x_2, \dots, x_n$ and multiplies the by the weight values $w_1, w_2, \dots, w_n$, then it calculates the sum of these h and compares it with a threshold value. If h is higher than the threshold value the neuron return 1 otherwise it returns 0. This functionality is mathematically expressed as follows:

$$y = f\left(\sum_{j=1}^{n} w_j x_j - u\right)$$

where $f(°)$ is the step function or some nonlinear approximation of it and u is the threshold value for the neuron. The $f(°)$ function is known as the *activation function* of the neuron. For simplicity we sometimes use the threshold u as another weight value such that $w_0 = -u$ which is fed into the neuron with a constant input of $x_0 = 1$. In that case the above equation can be rewritten as follows:

$$y = f\left(\sum_{j=0}^{n} w_j x_j\right), x_0 = 1$$

McCulloh and Pitts have proven that in general, appropriately selected weights result in neurons that are able to execute complete calculations. Compared to a biological system, this artificial neuron constitutes an oversimplification; however it has been generalized in many ways to produce some very potent systems. One obvious way to improve on this neuron is to use a different activation function such as the sigmoid function. The sigmoid function is defined as follows:

$$f(x) = \frac{1}{1 + e^{-ax}}, a > 0$$

where a is known as the rate of steepness of the curve. Usually a=1 but for different values of a the curve of the function is different.

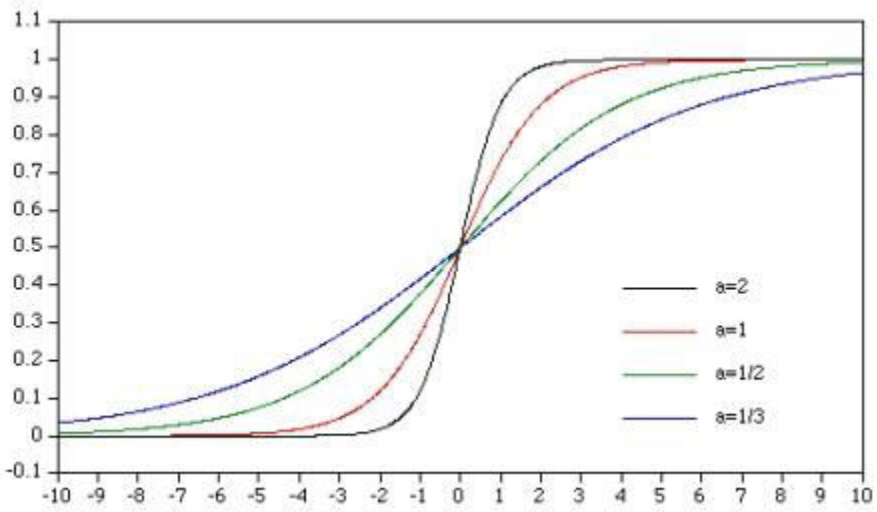


Figure 7: The sigmoid function's curve for different values of a.

After selecting an appropriate activation function we can train a neuron by appropriately changing the values of the weights of the inputs. A single perceptron, as it has been defined is able to only separate two classes. The training process for a single perceptron is done using the following steps:

First we choose a single data point out of the training data set. Next we calculate the response of the perceptron and we compare it with the correct answer for the selected data point. If the answer given by the perceptron is wrong then we adjust the weights according to the following equation:

$$\Delta w_i = \eta t_i x_i$$

where t_i is the desired response for the data point and η is the learning rate. This term can be used for the appropriate adjustment of the threshold value of the perceptron. The learning rate can

either be static or it can change in response to the output error. A learning rate that is proportional to the output error lead to a faster convergence of a neural network but it can also lead to unstable learning behavior.

A single perceptron however, is greatly limited in its ability to create intelligent systems. A most comprehensive proof of that is its inability to simulate the function of the exclusive-OR function. The reason for that is because the data points on the exclusive-OR function are not linearly separable. A layer of *n* perceptron can separate *n* different classes, as long as they are linearly separable. Researchers Minsky and Papert discovered that neural networks with multiple layers of perceptrons and nonlinear activation functions could simulate non linearly separable data.

Multilayer Perceptron

It is known that adding a layer of neurons combined with nonlinear activation functions can result in a neural network capable of separating non linearly separable classes of data. Therefore, such a network is able to depict k-dimensional vectors in a m-dimensional space with very small output errors. That type of neural network is known as a Multiplayer Perceptron and it can have any number of layers of perceptrons with any number of perceptrons in each layer. The first layer is known as the input layer and the last layer as the output layer. The layers between the input layer and the output layer are known as hidden layers. The question that needs to be addressed for a multilayer perceptron is how many hidden layers are required to succeed in the task it is designed for and what kind of activation functions do the neurons require.

The answer to this question has been given by researchers in the 1980s. In short, in a neural network based on perceptrons, one hidden layer is enough as long as it includes an appropriate number of neurons where the neurons use non constant and non-decreasing activation functions. This however sometimes results to a large number of neurons in the hidden layer which leads to some researchers to using an additional hidden layer such that the maximum number of total neuron does not exceed a certain limit. This is because the architecture of a neural network significantly affects the complexity of the network and the required training time.

Training rules

The lack of training algorithms for neural networks with many layers made their use impossible. That changed with the invention of training algorithms with the ability to train multiple layers.

Hebian rule

This is one of the first attempts at defining a widely used training rule for neural networks. This rule relies on changing the value of synaptic connections. More specifically, it proposes the reinforcement of the presynaptic and postsynaptic units when both of them were simultaneously activated. The basis for this adjustment is still being used in neural networks today. Given a neuron n_i that is connected to a neuron n_j with a connection weight of value w_{ij} and n_i is a positive input of value x_j produces a positive activation of value y_i then that connections weight is adjusted as follows:

$$w_{ij} = w_{ij} + \Delta w_{ij}$$

$$\Delta w_{ij} = \eta y_i x_j$$

where η is the positive learning rate.

Perceptron rule

A common type of neural network is one with m neurons in the input layer and a single neuron on the output layer. The training of such a neural network is done using supervision and the adjustment of the weights is done similarly to the Hebian rule where the change in weight value Δw_{ij} is defined as follows:

$$\Delta w_{ij} = \eta(t_i - y_i)x_j$$

where $y_i = f(net_i)$ with $f(°)$ is the activation function of neuron i and t_i is the desired response for neuron i. Just like in the Hebian rule the learning rate is positive, $\eta > 0$ but in the Perceptron rule the error is multiplied by the input x_i. This rule relies on supervision as is clear from the use of the desired response t_i. The Perceptron rule is a more powerful learning method than Hebian rule. On top of that, when the response is equal to the desired output, then the adjustment of the weights is zero.

Rosenblatt proved that if a set of weight values exists such that the perceptron gives the correct response to all training data, then the learning method that is based on the Perceptron rule will converge to that set of weight values in a finite number of steps. This is called the *perceptron convergence theorem*.

Delta rule

In 1959 researchers Widrow and Hoff developed neural network models which they named *Adaptive Linear Elements* and *Multiple Adaptive Linear Elements*. These models are two layer neural networks that, given the appropriate adjustment of their synaptic weights, can minimize the Mean Square Error. This algorithm of minimizing the Mean Square Error is known as the Least Mean Square algorithm and it has seen wide spread application ever since its inception.

The Delta rule is a supervised learning method which adjusts the synaptic weights similarly to the Hebian rule where the change in weight value Δw_{ij} is defined as follows:

$$\Delta w_{ij} = \eta(t_i - net_i)x_j$$

where net_i is the sum of the inputs of neuron i and the activation function is not considered. This rule is called Delta rule because it uses the difference ("Delta") of the inputs and the desired output for the adjustment of the weights. The above equation can be a considered a special case of the Perceptron rule where the activation function is the identity function $f(z) = z$.

Generalized Delta rule

While the Delta rule uses local response signals to calculate the error, the Generalized Delta rule uses global response signals to minimize the global Mean Square Error. The adjustment of the weights for this rule is once again similar to the Hebian rule but the change in weight value Δw_{ij} is defined as follows:

$$\Delta w_{ij} = \eta[t_i - f(net_i)]f'(net_i)x_j$$

It is important to point out that the first term of Δw_{ij} is $t_i - f(net_i)$, the same as in the Perceptron rule ($f(net_i) = y_i$) which ensures that Δw_{ij} will have a small value when $y_i \rightarrow t_i$. In addition to that the value of Δw_{ij} will be small when the derivative of the activation function is close to zero. The study of the derivative of the sigmoid activation function indicates that it is always positive and becomes close to zero as net_i becomes larger. This helps us ensure the stability of the changes in the parameters of the neural network, so that we don't have them fluctuating. This training rule is the basis for the backpropagation algorithm which can be used to tackle a variety of problems. There are however three major drawbacks of this training rule. If during training some of the weight values become too large, then the values of the corresponding derivatives will tend to be close to zero which results in zero adjustment of the weights, essentially halting the progress of optimization. Another drawback of the generalized Delta rule is that, just like all nonlinear gradient based optimization techniques, it can get stuck on a local minimum instead of finding the global minimum. Another drawback that also appears in other gradient optimization techniques is the

potentially slow convergence which increases the training time. A smaller learning rate can sometimes help remedy this, which demonstrates the value of a variant learning rate during training.

Kohonen rule

This unsupervised training rule is used in Self Organizing neural networks and relies on a technique called *competitive learning*. A neural network that uses competitive learning is a network in which every group or cluster of neurons competes for the right to be activated. This is complemented by determining an additional criterion that will force it to choose which neurons will be activated and which not.

The simplest neural network of this type consists of a single layer of neurons each of which is fully connected with the inputs. A typical neuron layer can be considered as a 2-dimensional self-organizing topographic feature map. In such neural networks, the position of the neurons, that have been stimulated the most with certain input data points, are related with patterns in those data points. The same goes for neighboring neurons that get stimulated from different input data points.

Researcher Teuvo Kohonen is the one that developed the widely spread self-organizing neural network that was named after him, Kohonen's Self-Organizing Feature Map (SOFM). This neural network adjusts its connection weights based only on the features of the input data points. Kohonen invented a training rule that can be used in various states of competitive learning in a way that forces the neural network to self-organize by selecting the most representative neurons. The most representative neuron is the one whose weights are closest to the input data point; this neuron is known as *winner neuron*.

The most extreme competitive learning strategy is the one according to which only the synaptic weights of the winner neuron are the ones that are represented. This kind of competitive learning requires that the weights of each neuron have been randomly instantiated. If the current normalized input data point is the *m*-dimensional vector *x* and there are q neurons in the layer of output neurons with p is the winner neuron then the weights of neuron p can be calculated as follows:

$$w_p^T = \max\{w_1^T x, w_2^T x, \dots, w_q^T x\}$$

For a positive learning rate η the adjustment of the weights is defined as follows:

$$w_{pj} = w_{pj} + \eta \Delta w_{pj}$$

$$\Delta w_{pj} = x_j - w_{pj}$$

This corresponds to the adjustment of only vector w_p respectively with a quantity that is equal to the difference between the current input vector and the current weight vector. No activation function is required to do that. After this adjustment is made, the weights neuron p tend to calculate the input vector better. Unfortunately, neurons with weight vectors that are really far apart from any input vectors can never win in the neuron competition and thus they cannot learn. Different variations in the training rule can solve this problem.

Less extreme variations of this strategy allow neighboring neurons to the winner neuron to adjust their weights. Determining the neighboring neurons can be done with the help of the Gaussian function which is defined as follows:

$$N_{p,j} = \eta(t) exp\left(-\frac{\|r_i - r_p\|^2}{2\sigma^2(t)}\right)$$

where p is the winner neuron, $\eta(t)$ is the learning rate which gradually decreases from iteration to iteration, r_i and r_p are the position vectors in the grid of output neurons for neurons i and p and $\sigma(t)$ is the width of the neighborhood which gradually decreases from iteration to iteration.

Alternatively, we can choose a specific geometry to determine the neighborhood neurons. The Euclidean distance is commonly used for this purpose such that the neighborhood contains all the neurons within a predetermined distance. Another popular geometry is a hexagonal neighborhood.

Backpropagation algorithm

The backpropagation algorithm is a training algorithm used in neural networks with many layers of neurons and its purpose is the minimization of the Mean Square Error. According to this algorithm, the error is propagated backwards into the network in order to adjust all the weights of the connections. Essentially this algorithm is a *steepest descent* method that minimizes the error of the desired outputs and the real outputs.

In order to decrease the error, this algorithm examines the slope of the error function and then determines the direction of the highest slope. This direction is used to minimize the error. The algorithm selects the negative of this slope, which describes the direction of the steepest descent. The error function's curve can be conceptualized as a bumpy surface, with peaks and valleys of varying width. If we start at a random point on the curve and we select direction of the negative slope in each iteration, we eventually end up in a valley. Therefore, we can't know of the valley is on a local minimum or the global minimum. Converging to a local minimum is a known issue of steepest descent methods.

The error function that the backpropagation algorithm minimizes is the Mean Square Error which is defined as follows:

$$E_p = \frac{1}{2}\sum_{i=1}^{N}(t_i - y_i)^2$$

This gives us the Mean Square Error of the neural network with respect to data point p where N is the number of neurons in the network, t_i is the desired output for neuron i and y_i is the real output for the neuron i. Consequently, the Mean Square Error for all the data points in the training dataset is defined as:

$$E = \sum_{p=1}^{N_p} E_p$$

Based on the theory that has been made on deepest descent methods and according to the generalized Delta rule, every weight of the neural network can adjust its value using the following equation:

$$w_{ij}(n+1) = w_{ij}(n) + \Delta w_{ij}(n)$$

$$\Delta w_{ij}(n) = -\eta \frac{\partial E}{\partial w_{ij}(n)}$$

where η is the positive learning rate and n is the number of iterations. This training rule requires the calculation of the derivative of the error function $\frac{\partial E}{\partial w_{ij}(n)}$ with respect to every weight w_{ij} of the neural network. For any certain neuron of a hidden layer, the output H_j is a nonlinear function f of the weighted sum of all the inputs of that neuron. Mathematically it can be expressed as follows:

$$H_j = f(net_j)$$

where f is the activation function of the neuron. The most widely used activation function is the sigmoid with a steepness rate of a=1 which is defined as follows:

$$f(x) = \frac{1}{1 + e^{-x}}$$

Using the *chain rule* we can write the derivative of the error function with respect to any weight in the neural network as follows:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial net_j} * \frac{\partial net_j}{\partial w_{ij}}$$

On top of that, given that the following is true:

$$net_j = \sum_{j=1}^{n} w_{ij} I_i$$

where I_i is the aggregate input of neuron *i,* we have that:

$$\frac{\partial net_j}{\partial w_{ij}} = I_i$$

Therefore, the initial equation can be rewritten as follows:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial net_j} I_i$$

$$\frac{\partial E}{\partial net_j} = \sum_{k=1}^{n} \frac{\partial E}{\partial net_k} \frac{\partial net_k}{\partial H_j} \frac{\partial H_j}{\partial net_j}$$

where *n* is the number of neurons and net_k is defined as follows:

$$net_k = \sum_{j=1}^{n} w_{ij} H_j$$

In addition:

$$\frac{\partial H_j}{\partial net_j} = f'(net_j)$$

Ultimately, for the derivative of the error function with respect to every network response, we have the following equation:

$$\frac{\partial E}{\partial net_j} = f'(net_j) \sum_{k=1}^{n} \frac{\partial E}{\partial net_k} w_{ij}$$

It has been proven that the derivative of the sigmoid function for a=1 is can be calculated as follows:

$$f'(net_j) = y_i(1 - y_i)$$

These equations can be used to calculate the error function's derivative with respect to any network response for the hidden layers of the network. For the output layer we have:

$$\frac{\partial E}{\partial net_j} = \frac{\partial E}{\partial H_j} f'(net_j)$$

where $\frac{\partial E}{\partial H_j}$ represents the difference between the real and the desired output and is therefore calculated as follows:

$$\frac{\partial E}{\partial H_j} = -(t_i - y_i)$$

To put all that into simpler terms, initially we calculate every neuron's response y_i for a given data point. The derivative of the error function with respect to all the weights in the network that the gradient descent algorithm requires is calculated as follows:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial net_j}\frac{\partial net_j}{\partial w_{ij}}$$

If *j* is an output neuron we have:

$$\frac{\partial E}{\partial net_j} = -(t_i - y_i)y_i(1 - y_i)$$

Otherwise, if the *j* is a hidden neuron we have:

$$\frac{\partial E}{\partial net_j} = (1 - y_i)\sum_{k=1}^{n}\frac{\partial E}{\partial net_k}w_{jk}$$

Finally the weights get adjusted as we mention with the following equation:

$$w_{ij}(n+1) = w_{ij}(n) + \Delta w_{ij}(n)$$

There are many variation of the basic algorithm that have been proposed to speed up the convergence of the neural network. The convergence is defined as the decrease of the overall Mean Square Error under a predetermined threshold. At this point the network is considered fully trained. A common variation is the addition of *momentum*, a term that regulates the effect of the previous adjustments in the weights on the current adjustment. This variation is defined as follows:

$$w_{ij}(n+1) = w_{ij}(n) - \eta\frac{\partial E}{\partial w_{ij}(n)} + a\Delta w_{ij}(n)$$

where η is the learning rate, usually 0.25 and a is the constant for the momentum.

Another variation of the basic algorithm that is used to speed up the convergence is the addition of random noise. It has been proved that while inaccuracies due to digital quantization are detrimental to the convergence, analog noise can actually help speed up the convergence.

The backpropagation algorithm's ability to train a neural network with multiple layers has allowed to become wide spread and find success in a variety of applications. It is not however without its flaws, mainly related to the speed of convergence as well as the complexity of the algorithm which prevents it from being used on hardware.

Quickprop

Another variation of the backpropagation algorithm uses the second derivative of the error function without calculating the Hessian matrix. It requires a copy of the previous slope vector of the error function as well as the previous adjustments of the weights. Quickprop is an attempt to reduce the number of iterations required for convergence and it achieves that by an order of 5 over the original backpropagation algorithm. The weights are adjusted the same as the original backpropagation but the derivative of the error function with respect to each weight is defined as follows:

$$\Delta w_{ij}(n) = \Delta w_{ij}(n-1)\left(\frac{\nabla E_{ij}(n)}{\nabla E_{ij}(n-1) - \nabla E_{ij}(n)}\right)$$

Kohonen's Self-Organizing Feature Map (SOFM)

This neural network is an unsupervised learning network. It assumes a vector space *X* of dimension *p* with a finite number of *n* vectors which can be defined as follows:

$$X_k = [x_{0,k}, x_{1,k}, \dots, x_{p-1,k}]^T, k = 1, \dots, n$$

Let's also assume a network of neurons *c*, grouped up either as a 2-dimensional grid or a one dimensional grid. The grid is fed *p*-dimensional input vectors from a common source of *p* neurons where the p features $x_{0,k}, x_{1,k}, \dots, x_{p-1,k}$ of the input vector X_k are all fed into the neurons of the grid. The connection of a neuron *j* with the *p* features of the input vectors is done using *p* separate synaptic connections. The synaptic connections of neuron *j* are represented by their weight values which can be described as a vector as follows:

$$W_j = \left[w_{j,0}, w_{j,1}, \dots, w_{j,p-1}\right]^T, j = 0,1, \dots, c-1$$

There are two empirical rules that are applied, one for the learning rate and the other for the width of the neighborhood for each neuron in the network.

The first rule states that the learning rate is the same across the network and it decreases monotonously as training progresses.

The second rule states that the width of the neighborhood for each neuron decreases monotonously as training progresses.

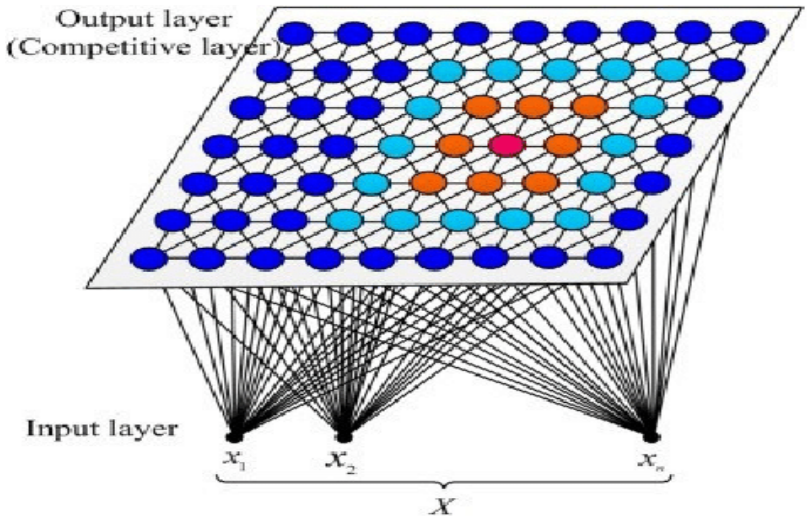


Figure 8: The structure of a Self-Organizing Feature Map neural network.

The positions of the neurons on the grid are constant and do not change during training and the same goes for the connections between the neurons. The unchanging topology of the network allows it to cluster similar input vectors to neighboring neurons. This however is not always the case, for cases where the input vector is of greater dimension than the neural network's grid, in which case the network cannot describe the topology of the vector and so for two similar input vectors X_a, X_b, they would be clustered to neurons that are far away from each other.

For the training of the neural network we define a time parameter t which increases, each time a new input vector is fed into the network.

The training process is done by serially inputting vectors from the input vector space into the neural network. Every λ consecutive input vectors define an *epoch* and the total number of elapsed epochs is defined by the variable *ep*.

For every input vector X_k that belongs in the input vector space, we define a winner neuron as the neuron whose weight vector is the closest to the input vector. For the calculation of the distance between vectors we can use the Euclidean distance or the Manhattan distance.

The function $nei(j, ep)$ defines the set of neurons that belong in the neighborhood of neuron j after the passing of ep epochs.

As the number of epochs goes up, the radius of the winner neuron decreases until it becomes zero, which gradually limits the effect of the winner neuron on its neighbors. The simplest way for decreasing the neighborhood radius is linearly. A good starting distance is usually two thirds of the length of the longest side of the grid and the decrease to zero should be done in about 150 epochs.

An important factor of the speed of convergence as well accuracy of the results is the learning rate $\varepsilon(ep)$. A standard starting value for the learning rate is 0.05 and a transition to a final of 0.0005 within 150 epochs is also standard. When the learning rate is too high the weight vectors of the neurons which define the centers of the clusters approximate the input vectors with a high speed which increases the risk of fluctuating weight vectors. When the learning rate is too low the weight vectors get stabilized which makes convergence faster but increases the risk of not converging to an optimal solution. The gradual transition between a high learning rate to a low learning rate eliminates both these risks.

In order to train Kohonen's Self-Organizing Feature Map we need to go through the following steps:

First we define the desired number of clusters c, the topology of the network and the way that a neuron's neighborhood is calculated. Next the weight vectors of all the neurons get instantiated with randomly selected vectors out of the input vector space. The variables t and ep are initialized to zero. Next we select an input vector X_k from the input vector space and we calculate the winner neuron w_1 which satisfies the following criterion:

$$\|X_k - W_{w_1}\| \leq \|X_k - W_j\|, \forall\, j = 0,1, \dots, c-1$$

Next we find the group of neurons that belong in the neuron's neighborhood which is represented as $nei(w_1, ep)$. We also calculate the learning rate $\varepsilon(ep)$. We adjust the weight vector of the winner neuron as follows:

$$W_{w_1}(t+1) = W_{w_1} - \varepsilon(ep)(X_k - W_{w_1})$$

Adjust the weight vectors of the neurons that belong in the neighborhood of the winner neuron as follows:

$$W_{w_1}(t+1) = W_m - \varepsilon(ep)(X_k - W_m), \forall m \in nei(w_1, ep)$$

We check if there has been λ input vectors fed into the network in which case increase the epoch by 1. After each epoch is elapsed, we check if the weight vectors $w_{j,0}, w_{j,1}, \dots, w_{j,p-1}$ of all the neurons $j = 0,1, \dots, c-1$ have changed substantially compared to the previous epoch. If the change of the weights between epochs is smaller than a predetermined threshold, then the network is considered to be converged and the training stops. In addition we check if the maximum number of epochs ep_{max} has been elapsed and if it has then the network has been given enough time to converge and the training stops. If none of the above criteria are met then a new input vector is selected and the process starts over.

Learning Vector quantization (LVQ)

This neural network was initially designed by researchers Linde and Gray for image compression but Kohonen modified it for pattern recognition. The Learning Vector Quantization neural network is thus considered to be an extension of Kohonen's SOFM neural network with a supervised learning method. It combines the simplicity of a Self-Organizing neural network with the accuracy of supervised learning. LVQ is a neural network that relies on the training of competing layers with a supervised fashion. As we have mentioned, a competitive layer of neurons is able to learn by itself and to cluster the input vectors. Consequently, the clusters that a competitive layer comes up with are based entirely on the distance between the input vectors. If two input vectors are close to each other they will probably be clustered into the same cluster. In a strict competitive learning system, there is no mechanism that determines whether or not any input vector truly belongs in a cluster. The LVQ neural network however, learns to cluster the input vectors to classes that are predetermined by the user.

In the LVQ network the input vector space is separated into distinct regions and each region is represented by a single vector. The set of all these vectors is called *codebook* and each of the vectors is called *code vectors*. The LVQ network clusters each input vector by comparing it to each vector in the codebook and assigns the vector to the cluster whose code vector is closest according to the Euclidean distance.

The LVQ neural network is consisted of an input layer and an output layer and the competitive layer is located between them. The competitive layer needs to contain enough neurons such that each cluster is represented by a sufficient neuron quantity.

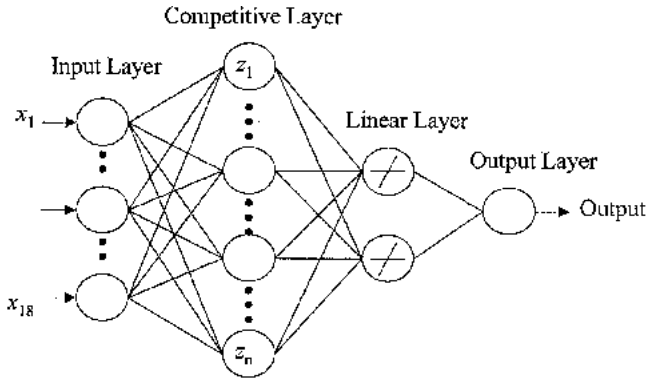


Figure 9: The structure of an Learning Vector Quantization neural network.

The LVQ learning algorithm consists of two stages. In the first stage, an unsupervised clustering technique, usually Kohonen's SOFM, is used to determine the number of clusters and the center of each cluster is used to initialize the code vectors. At this point, each input vector is matched to a

label of the SOFM's clustering of that input vector. In the second stage, the clustering information form the initial clustering is used to fine-tune the centers of the clusters. The goal at this point is to construct a complete codebook of vectors that optimally represent each cluster.

In order to train an LVQ neural network we need to go through the following steps:

First we initialize the weight vectors and construct the initial codebook which can be done using Kohonen's SOFM as we described or alternatively K Means. We define the starting learning rate $a(t)$ where t=0, usually as 0.01 or 0.02. The learning rate decreases monotonously during training. We randomly select an input vector *x* and we calculate the winner neuron *c* as follows:

$$\|w_c(t) - x(t)\| \leq \|w_j(t) - x(t)\|, j = 1, \dots, n \; and \; c \neq j$$

If *x* belongs in the cluster of the winner neuron c, we adjust the weights of the winner neuron as follows:

$$w_c(t+1) = w_c(t) + a(t)[x(t) - w_c(t)]$$

Otherwise we adjust the weights as follows:

$$w_c(t+1) = w_c(t) - a(t)[x(t) - w_c(t)]$$

Next we decrease the learning rate and we check if any of the convergence criteria is met, such include the maximum number of iterations or a threshold for the learning rate. If any of these criteria are met, the network is considered to have converged and the training stops. If none of the criteria are met then we select a new input vector.

This process ensures that when an input vector belongs in the same cluster as the winner neuron, then that cluster's center comes closer to the input vector, otherwise it comes further away from it.

LVQ2 Algorithm

In this variation of the LVQ training algorithm, we don't only adjust the weights of the winner neuron but also the weights of the second winner neuron. The basic idea here is that if the first winner neuron doesn't represent the correct cluster but the second winner neuron does, it may be desirable to adjust the centers of both clusters. This adjustment however comes with the following prerequisites:

First, the winner neuron belongs in a different cluster from the input vector. Additionally, the second winner neuron belongs in the same cluster as the input vector. Lastly, the Euclidean distance between the input vector and the winner neuron as well as the second winner neuron,

d_c, d_r respectively, then the input vector needs to be located within a window of width ε which is defined as follows:

$$\min\left(\frac{d_c}{d_r}, \frac{d_r}{d_c}\right) > \frac{1-\varepsilon}{1+\varepsilon}$$

where $\varepsilon \in [0.2, 0.3]$. Alternatively the window can be defined as follows:

$$\left|\frac{d_c}{d_r} - 1\right| \leq \varepsilon$$

where ε depends on the number of training input vectors and a typical value is 0.35. If these criteria are met then the weight vectors are adjusted as follows:

$$w_c(t+1) = w_c(t) - a(t)[x(t) - w_c(t)]$$

$$w_r(t+1) = w_r(t) + a(t)[x(t) - w_r(t)]$$

LVQ3 Algorithm

This algorithm is an improvement on the LVQ2 algorithm with the addition of the potential adjustment of the weights of the first and second winner neuron when both belong in the same cluster as the input vector. In that case the adjustment becomes is defined as follows:

$$w_s(t+1) = w_s(t) + ma(t)[x(t) - w_s(t)]$$

where $s = c \; or \; r$ and $ma \in [0.1, 0.5]$ is known as the *relative learning rate*.

Convolutional Neural Networks

These neural networks are a special kind of network that is designed to process grid like data. This could include time series, which can be formatted as a 1-dimensional grid where time is the axis as well as image data which is essentially a 2-dimensional grid of pixels where width and height are the axes. Convolutional networks have been tremendously successful in practical applications. The name "convolutional neural network" indicates that the network uses convolutions.

Convolution is a linear mathematical operation that is applied on two functions of a real-valued argument. In the context of convolutional neural networks, the two functions would be the input to a neuron and the *kernel* of the convolution. For a 2-dimensional image *I* as input and a 2-dimensional kernel *K* the operation of convolution is defined as follows:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$

Convolution is commutative, meaning we can write the above definition as follows:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)$$

Usually the latter formula is more straightforward to implement, because there is less variation in the range of valid values of *m* and *n*. In practice, many neural network libraries implement a related function called the *cross-correlation*, which is the same as convolution but without flipping the kernel and is defined as follows:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n)$$

Cross-correlation and convolution in practice are essentially interchangeable. We can conceptualize convolution within the context of an image as "sliding" the kernel on the image and adding the products of corresponding elements. More specifically, a kernel of size *n*x*n* will be matched with a *n*x*n* window onto the image, then the corresponding elements will be multiplied and added to produce a single pixel value of the result. The resulting pixel value corresponds to the center element of the kernel. The same process happens for all the possible *n*x*n* windows on the image until the result is complete. Convolution leverages three important ideas that can help improve a machine learning system; *sparse interactions*, *parameter sharing* and *equivariant representations*.

Traditional neural network layers use matrix multiplication by a matrix of parameters with a separate parameter describing the interaction between each input unit and each output unit. This means that every output unit interacts with every input unit. Convolutional networks, however, typically have sparse interactions (also referred to as sparse connectivity or sparse weights). This is accomplished by making the kernel smaller than the input. For example, when processing an image, the input image might have thousands or millions of pixels, but we can detect small, meaningful features such as edges with kernels that occupy only tens or hundreds of pixels. This means that we need to store parameters, reducing the required storage as well as the number of operations required and consequently the complexity of the solution. These improvements in complexity become substantial when we tackle tasks with a large number of high resolution images.

Parameter sharing is the property of a model to use the same parameters for multiple functions. In the case of convolutional neural networks, the kernel is not used only once but at every position of the input image with the exception of the bounding pixels. For instance, if we wanted to detect the edges of an image, we would use the same parameters across the entire image. This attribute ensures that the network does not have to learn separate parameters for every location but rather a single set of parameters. This does not affect the time complexity of the network but it does further reduce the storage requirements since the kernel size is usually orders of magnitude smaller than

the input images. In short, convolution is far more effective than dense matrix multiplication for use in neural networks.

Equivariant representations or *equivariance* is a property of the neural network that is caused by the kind of parameter sharing it employs. For a function that has this property, if the input of the function changes, then the output of the function changes in the exact same way. Mathematically we would describe this as follows:

$$f\big(g(x)\big) = g(f(x))$$

In the case of convolution, if we apply a function g to an image and then perform convolution it would be the same as performing convolution and then applying the function. When processing time-series data, this means that convolution produces a sort of timeline that shows when different features appear in the input. A shift in the time of any input will not affect its representation of the input, only the time its output appears. In the case of images, certain features get extracted from certain locations of the input image and if we move the object on the input image, the feature that will be extracted will have moved in the same way. This property is useful in cases where we can benefit from applying a function to several locations in an image. Convolution however is not equivariant for some functions such as scaling and rotation. These functions require special handling.

Using convolution in a neural network usually requires three stages. First the inputs are fed into a layer of neurons which perform several convolutions in parallel to produce a set of linear activations. Then these activations are run through a non-linear activation function, usually *rectified linear* or *leaky rectified linear*. In the last stage a special operation called *pooling* is used to modify the output of the layer.

Pooling is a function that replaces the output of a layer with a summary statistic of the layer's outputs such as maximum. *Maxpooling* for instance replaces the output of a layer with the maximum output within a rectangular neighborhood. Other pooling functions apply different operations to the outputs such as the average of a rectangular neighborhood, L^2 norm of a rectangular rectangle or the weighted average of a rectangular rectangle using the distance from the center of the rectangle. Pooling is a useful technique to make the outputs of a layer approximately invariant to small translations in the input. This means that if we were to translate the input by a small amount the pooled outputs will not change. This is an important attribute to instill to a network if we care that a certain feature exists but necessarily its exact location on the image. For example, in face recognition we don't care about the exact location of the eyes but we do

care that the network is able to detect the existence of eyes. This attribute is not always desirable so the decision on how to use pooling relies on the task we are trying to solve.

Pooling over spatial regions produces invariance to translation, but if we pool over the outputs of separately parameterized convolutions, the features can learn which transformations to become invariant to. Pooling is essential in tasks where the input images do not have a fixed size. In that case, in order to classify the images of varying size, we use a varying offset between the pooling regions so that the classification layer always receives the same number of summary statistics about the image regardless of the input size.

When training neural networks we need to be careful to avoid the problem of overfitting. Overfitting is a situation where after training, the weights of the network are so tuned to the training examples they are given that the network doesn't perform well when given new examples.

In order to avoid that problem in a convolutional neural network, it is common to use a certain operation that is called *dropout*. The idea of dropout is simplistic in nature. This layer "drops out" a random set of activations in that layer by setting them to zero. The benefit of such a simple and seemingly unnecessary and counterintuitive process is that in a way, it forces the network to be redundant. This means the network should be able to provide the right classification or output for a specific example even if some of the activations are dropped out. It makes sure that the network isn't getting too "fitted" to the training data and thus helps alleviate the overfitting problem. An important note is that this layer is only used during training, and not during test time.

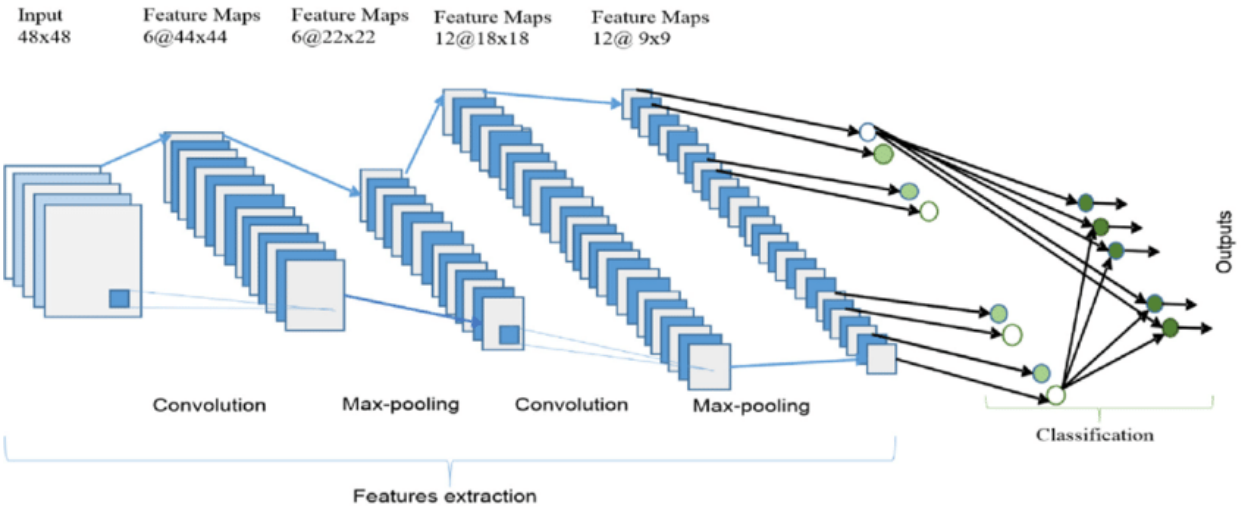


Figure 10: The structure of a Convolutional Neural Network.

Other operations besides convolution are usually necessary to implement a convolutional network. To perform learning, we need to be able to calculate the derivative of the error function with

respect to the kernel, given the error of the outputs. In some simple cases, this operation can be performed using the convolution operation, but many of the more interest cases, including the case do not have this property.

Since convolution is a linear operation, consequently it can be described as a matrix multiplication. The matrix involved is a function of the convolution kernel. The matrix is sparse, and each element of the kernel is copied to several elements of the matrix. This view helps us to derive some of the other operations needed to implement a convolutional network. The operation that is needed to backpropagate error derivatives into the network is multiplication by the transpose of the sparse matrix defined using convolution, thus performing this operation is essential to training neural networks with more than one hidden layers.

Another essential operation for training convolutional neural networks is the input gradient operation which can sometimes be implemented using a convolution but in general requires a third operation to be implemented is necessary for the model to work properly. Coordinating this transpose operation with the forward propagation is necessary for the neural network to work properly. The size of the output that the transpose operation should return depends on whether or not we have zero-padding, the size of stride of the forward propagation operation, as well as the size of the forward propagation's output map. In some cases, multiple sizes of input to forward propagation can result in the same size of output map, so the transpose operation must be explicitly told what the size of the original input was.

These three operations, convolution, backpropagation from output to weights, and backpropagation from output to inputs, are sufficient to calculate all the derivatives of the error function with respect to any parameter of the network and to train any layer of feed forward convolutional network, as well as to train convolutional networks with reconstruction functions based on the transpose of convolution.

The use of Deep Convolutional Neural Networks (DCNN) has pushed the progress of computer vision to an unprecedented level. Image classification and object detection have been among the biggest successes in computer vision, as state of the art classifiers are used daily for a variety of complicated tasks such as medical image recognition, face detection, product classification among many. This progress has been possible thanks to the modern DCNN architectures, which are able to extract image features of superior quality than any hand crafted features thus achieving very good results.

Image segmentation is too one of the computer vision problems that has seen great progress with the use of DCNNs, as new state of the art neural networks progressively push the accuracy and performance on some of the hardest problems.

In recent year, the rise of Deep Convolutional Neural Networks (DCNN) has improved significantly the feature quality, with state of the art neural networks such as ResNet [2], DenseNet [4], Inception [5], GoogleNet [6] and VGGNet [3] among others, for image classification and Mask R-CNN [7], U-Net [8] and DeepLab [1] for image segmentation.

These state of the art neural networks have introduced some innovative architectures for feature extraction, as well as some interesting variations that can be implemented to achieve the best possible result depending on the task and the available data. Most of these neural networks use several fully connected convolutional layers, with features extracted from lower levels of the network being fed forward deeper into the network. The convolutional layers are usually followed by a pooling layer, as is the case with VGGNet, ResNet and others. After the pooling layer several fully connected convolutional layers follow and the process continues. This process allows for the extraction of progressively more expressive features which are ultimately fed into a linear classifier and a softmax layer. A particularly interesting variation of this architecture is introduced by DeepLab which uses *Atrous Convolution* to improve on the feature density of the extracted features.

Clustering algorithms have also been used to perform image segmentation on image data in cases where the task does not specify a certain number of classes, in which case a classifier would be ideal. The K-Means algorithm as well as the Expectation Maximization algorithm has been used on raw image data with relative success, especially in segmentation of medical images.

Using features extracted from deep neural networks for a different task is not a new idea. In fact, extract features from the fully connected convolutional layers of the state of the art neural networks, in combination with a custom linear classifier has wielded great results in tasks where the available data was limited. Features extracted from neural networks have also been used for clustering image data in combination with popular clustering algorithms.

Data Augmentation

Approaches that alter the training data in ways that change the array representation while keeping the classification label the same are known as data augmentation techniques. Data augmentation techniques provide ways that we can make our existing dataset even larger, just with a couple easy transformations. Some popular augmentations people use are grayscale representations, horizontal

flips, vertical flips, random crops, color jitters, translations, rotations, and much more. By applying just a couple of these transformations to our training data, we can easily double or triple the number of training examples.

Gaussian Mixture Model

In statistics a mixture model is a probability model that describes a set of data populations as a singular population. A mixture model is therefore defined by a single probability distribution that incorporates the probability distributions that define all its sub-populations. In essence a mixture model is a probability model that expresses the properties of data with different probability distributions.

That allows a mixture model to describe data with more complex distributions and which makes it good for modeling real world data. A mixture model is defined by a set of K probability distributions which are described by the set of parameters that uniquely define each of the K probability distributions. A mixture model can optionally include a set of K weights, the sum of which is 1, such that each corresponds to a distribution and defines its contribution to the mixture model's distribution. Any random variable that is described by a mixture model of K distributions is required to have K components.

Mixture models are usually used to do deduce properties of the K distributions by using random variables that are described by the mixture model's distribution. In the context of pattern recognition, mixture models can be used for clustering and unsupervised learning.

The simplest mixture model is the Gaussian Mixture Model which is defined by a set of K Gaussian or Normal distributions that may be univariate or multivariate and a set of K weights. A Gaussian Mixture Model requires a mean for each of its K distributions but in order to convey the relations between the distributions it also requires the covariance between the variables of each distribution. All the covariances are described as a square K x K positive defined matrix called covariance matrix. A Gaussian Mixture Model is therefore uniquely defined by its K means, covariance matrices and weights.

To determine the optimal values for the model we need to find the maximum likelihood of the model. We can find the likelihood of the model by calculating the joint probability of all the N data points, which is defined as:

$$p(X;\pi,\mu,\Sigma)=\prod_{n=1}^{N} p\left(x^n;\pi,\mu,\Sigma\right) \quad (1.1)$$

$$p(x^n;\pi,\mu,\Sigma)=\sum_{j=1}^{k} \pi_j N\left(x^n;\mu_j,\Sigma_j\right) \quad (1.2)$$

$$N\left(x^n;\mu_j,\Sigma_j\right)=\frac{1}{\sqrt{(2\pi)^d\left|\Sigma_j\right|}}\cdot e^{-\left(x^n-\mu_j\right)^T\Sigma_j^{-1}\left(x^n-\mu_j\right)/2} \quad (1.3)$$

Where X represents the N data points of a given dataset, π represents the set of K weights, μ represents the means of the K distributions, Σ represents the covariance matrices for the K distributions and the equation (1.3) represents the probability density function of a multivariate Gaussian distribution.

In order to determine the optimal values for the parameters of the model given the equation (1.1) we need to use an iterative method for optimization called Expectation Maximization also known as the EM algorithm.

The EM algorithm

The EM algorithm is an iterative method that is widely used to estimate the parameters of a statistical model in order to maximize the likelihood function of the model. It is essentially the alternation of two operations, *Expectation* and *Maximization*, for several iterations.

Initially the parameters of the statistical model need to be initialized. Usually the initial initialization of the parameters is random; occasionally however the parameters are initialized as the results of another iterative method such as KMeans. A good initialization is important because it means fewer iterations until convergence. In our case let the parameters of a Gaussian Mixture Model be:

$$\theta=\{\pi,\mu,\Sigma\} \quad (2.1)$$

The *Expectation* step calculates the likelihood function for the current values of the parameters of the model. In essence the likelihood calculated during the *Expectation* step defines the responsibility of each of the N data points with respect to the K distributions of the model. In other words the *Expectation* step calculates how likely it is for any data point to belong in a specific distribution.

The calculation made during the Expectation step, for the Gaussian Mixture Model for a given iteration *t* is the one that follows:

$$r_{nk}^{(t)} = \frac{\pi_k N(x^n; \mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j N(x^n; \mu_j, \Sigma_j)} \quad (2.2)$$

The *Maximization* step updates the parameters of the K distributions such that the likelihood function of the model is maximized. In order to do so the *Maximization* step uses the responsibilities calculated during the *Expectation* step for each of the N data points. The parameters that get updated for the Gaussian Mixture Model are the K means, covariance matrices and weights. The updated values for these parameters for a given iteration *t* are defined as follows:

$$\mu_k^{(t+1)} = \frac{\sum_{n=1}^{N} r_{nk}^{(t)} \cdot x^n}{\sum_{n=1}^{N} r_{nk}^{(t)}} \quad (2.3)$$

$$\Sigma_k^{(t+1)} = \frac{\sum_{n=1}^{N} r_{nk}^{(t)} \left(x^n - \mu_k^{(t+1)}\right)\left(x^n - \mu_k^{(t+1)}\right)^T}{\sum_{n=1}^{N} r_{nk}^{(t)}} \quad (2.4)$$

$$\pi_k^{(t+1)} = \frac{1}{N} \cdot \sum_{n=1}^{N} r_{nk}^{(t)} \quad (2.5)$$

Where $r_{nk}^{(t)}$ is the responsibility of data point *n* for the distribution *k* during iteration *t*. The means of the K distributions μ are updated as defined by the equation (2.3), the covariance matrices Σ as defined by (2.4) and the weights π as defined by (2.5).

The alternation of the *Expectation* and *Maximization* steps is repeated until some convergence criterion is met. For instance, when the changes that happen to the means of the distributions, after an update, are smaller than a specified tolerance level. The EM algorithm may not converge on the global maximum of the likelihood function but instead a local maximum, that is why a convergence criterion needs to be specified.

After the EM algorithm has converged, the parameters of the K distributions of the Gaussian Mixture Model will be updated optimally, so that the likelihood function of the model is maximized. At this point the Gaussian Mixture Model may be used for inference on new data.

We can do inference on new data points using the Gaussian Mixture Model by performing a single *Expectation* step on any new data point. This will give us the responsibility of the new data point with respect to each of the K distributions of the model. In other words we can use the *Expectation* step to calculate the probability that the new data point belongs in each of the K distributions. This means that we can use a Gaussian Mixture Model for soft clustering.

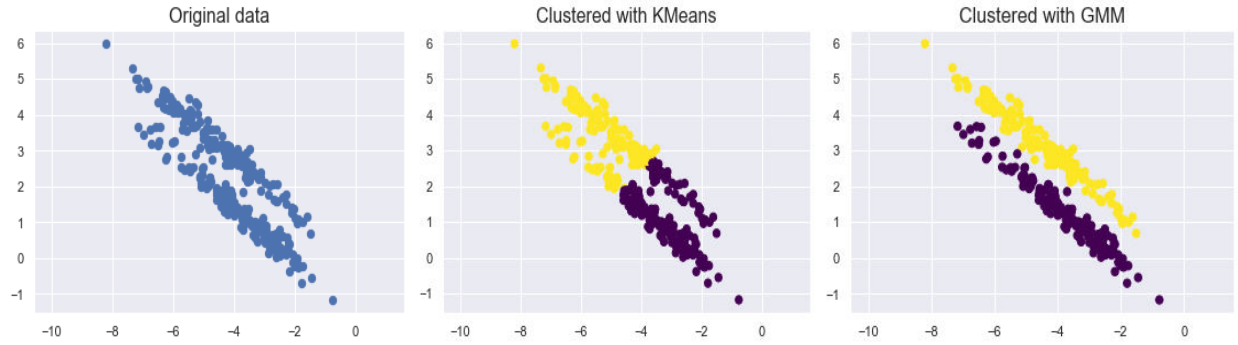


Figure 11: Example where soft clustering (Gaussian Mixture Model) performs better than hard clustering (K Means).

The reason that we used a Gaussian Mixture Model for the purposes of image segmentation is that we needed a clustering technique with the capability of modeling the complex feature data extracted from a neural network. The neural network we chose to extract features from is the DeepLabV3+ neural network.

DeepLabV3+

This neural network belongs in Tensor Flow's Model Garden. It has been trained for semantic image segmentation on the *PASCAL VOC-2012* dataset with 21 classes for segmentation. It is a *Deep Convolutional Neural Network* (DCNN) and it uses several high level methods to achieve the *state of the art* solution at the *PASCAL VOC-2012* problem at 2017, reaching 79.7% mean *Intersection over Union* (mIoU). One of the ways that the DeepLab network is special is that it uses *Atrous Convolution* or Dilated Convolution, a variation of the standard convolution that introduces a spacing between the elements of the convolution kernel, effectively increasing the field of view of the convolution without increasing the number of parameters in the convolution kernel. The operation for the *Atrous Convolution* for a 1D input array x is defined as:

$$y[i] = \sum_{k=1}^{K} x[i + r \cdot k] \cdot w[k]$$

Where w is the convolution kernel, K is the convolution kernel's length and r is the rate of dilation or the spacing between the kernel's elements. *Atrous Convolution* allows for the DeepLabV3+ network to retain high spatial resolution in its features. This means that the feature maps generated by the convolutional layers of the network are dense, allowing for more accurate segmentation.

Deep Features

The idea of extracting features from a state of the art neural network and using them in combination with other techniques is an extremely powerful idea. Instead of engineering features from scratch we can extract as many features as we need from neural networks trained on large datasets and use them to get good results on our own task using a smaller dataset. Some of the state of the art neural networks have been trained on huge and challenging datasets and by repurposing the features that they generate, we are able to exploit the knowledge that these neural networks were able to extract from these datasets.

However not all features are as useful; the features extracted from lower layers of the network are more general and less task specific compared to features from deeper layers of the network. In the context of semantic segmentation this means that features from deeper layers of the network have more significant semantic information. Therefore it makes sense to use features extracted from deeper layers, the more similar our task is with the original. Similarly if our task is entirely different from the task of the original neural network, then the information of features extracted from the deeper layers is going to be too high level and task specific and will not wield the best results.

To achieve the best results we need to extract features from intermediate layers of the neural network, deep enough so that the features contain as much useful information as possible, but not so deep that they are too task specific.

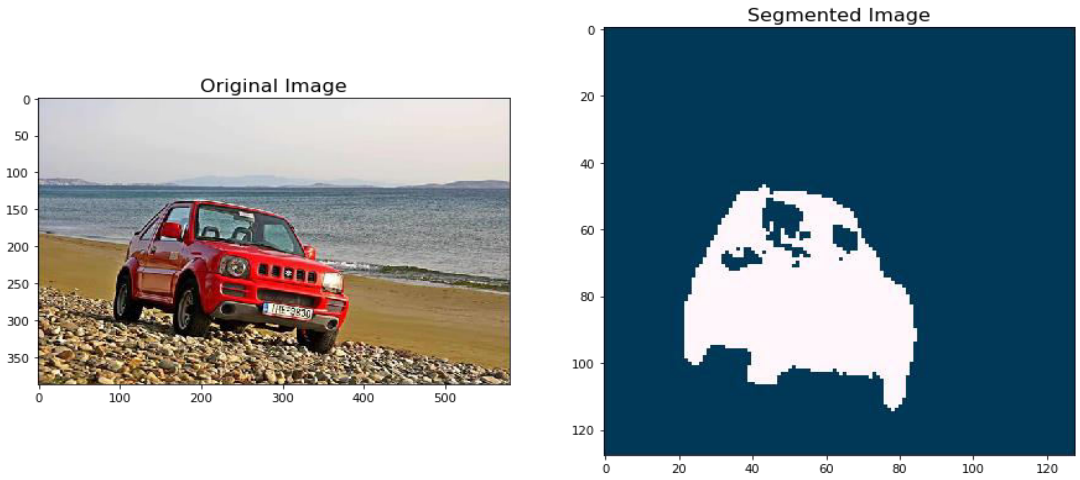


Figure 12: Original image and the resulting segmented image with a Gaussian Mixture Model of n=2 clusters. The segmented image has some noise due to imperfections of the segmentation.

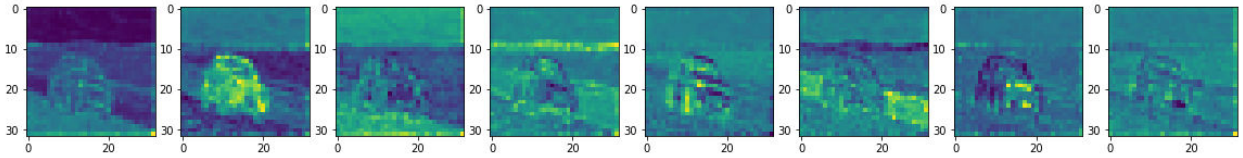


Figure 13: n = 8 *Principal Components* for features extracted from layer 67.

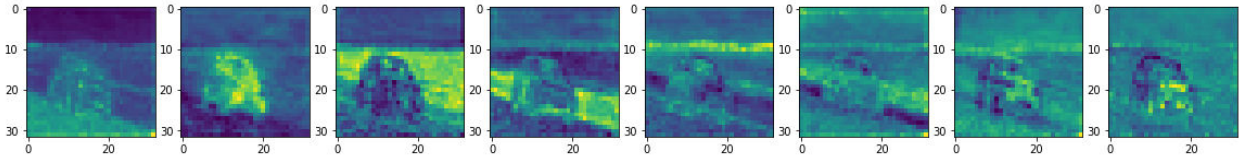


Figure 14: n = 8 *Principal Components* for features extracted from layer 131.

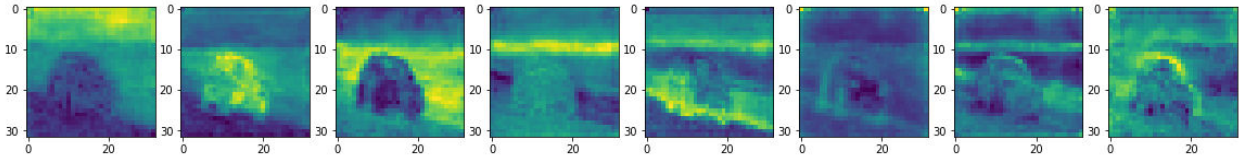


Figure 15: n = 8 *Principal Components* for features extracted from layer 259.

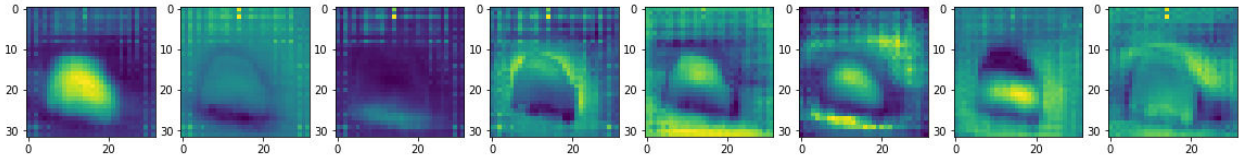


Figure 16: n = 8 *Principal Components* for features extracted from layer 355.

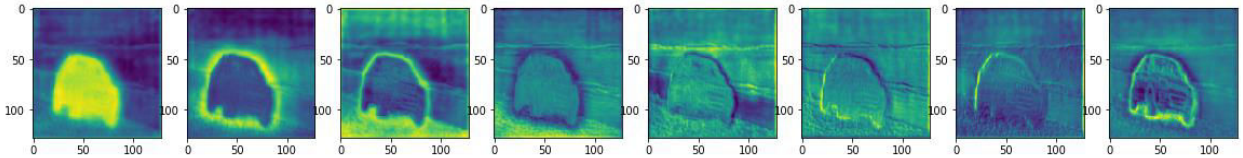


Figure 17: n = 8 *Principal Components* for features extracted from layer 405.

One of the most significant attributes of the extracted features is the high dimensionality. If we were to use these high dimension features to fit the parameters of a Gaussian Mixture Model, the result would certainly be affected by the *Curse of Dimensionality*.

The Curse of Dimensionality

The curse of dimensionality is an important problem that stands to limit the performance and accuracy of our model. It is an effect that always becomes a concern when we do pattern recognition on some data in high dimensional spaces. When we input high dimension data to a machine learning model, this data defines the space in which the model will search for the optimal

solution for the problem. So, if the data is high in dimension that means that the space which the model has to explore to find the solution is going to be really big. That makes it so the model requires more and more data to find a good solution. In fact, as the dimension of the data increases, the amount of data required increases exponentially and eventually it reaches the point of diminishing returns where inputting the model more data does not increase its accuracy and performance.

The reason for this effect happening is that when the dimension of the data increases, the data becomes sparse and it becomes difficult for pattern recognition algorithms to detect the similarities in the data. Therefore, more data is required but this only makes the problem worse. Also, more data and high dimension data means longer training times and higher memory requirements. It is therefore necessary to combat this effect by reducing the data dimension. A common technique for this is *Principal Component Analysis*.

Principal Component Analysis (PCA)

Principal Component Analysis is a process that allows us to transform a set of data such that the variance of each feature of the data is maximized. In other words *Principal Component Analysis* allows us to "reframe" the data in a way that its features are better separated.

A benefit of this is that we can order the features of the data by their variance, essentially how important each feature is to the entirety of the data. With that in mind we can reduce the dimension of the data by discarding the features with the lowest variance and keeping as many of the features with the highest variance as we need.

In order for PCA to achieve this, first the data needs to be normalized. This is an essential step to the process because if there are large differences between the ranges of the values of the data, those values will dominate over the other values leading to biased results. Normalizing the data is a simple transformation:

$$x' = \frac{x - \mu}{\sigma} \quad (3.1)$$

where x is the value of feature of the data μ is the mean of the value distribution for that feature of the data and σ is the standard deviation of the value distribution.

The next step is to calculate the covariance matrix for the features of the data. The covariance matrix is an N x N symmetric matrix, where N is the number of dimensions, and it contains the variance of each feature of the data with respect to all the other features. The covariance matrix can be defined as follows:

$$\begin{bmatrix} Cov(x_1, x_1) & \cdots & Cov(x_1, x_n) \\ \vdots & \ddots & \vdots \\ Cov(x_n, x_1) & \cdots & Cov(x_n, x_n) \end{bmatrix} \quad (3.2)$$

Where $x_1, .., x_n$ are the features of the data. Positive covariance between two features means they are correlated and negative covariance means they are inversely correlated. Since the covariance of a variable with itself is equal to its own variance ($Cov(x_n, x_n) = var(x_n, x_n)$), that means that all the elements of the diagonal are the variances for each feature of the data. On addition to that the covariance of two variables is commutative ($Cov(x, y) = Cov(y, x)$), which means that the covariance matrix is symmetric. The covariances are calculated as follows:

$$Cov(X, Y) = \frac{1}{n-1} \sum_{i=1}^{n} (X_i - \bar{x})(Y_i - \bar{y}) \quad (3.3)$$

Where X, Y are the discrete random variables that represent the features of the data, $\bar{x}$ and $\bar{y}$ are their respective means and X_i and Y_i for i = 1,…,n are the values they can take.

PCA uses the covariance matrix to create new features that are linear combinations of the original data features. These new features called *Principal Components* are orthogonal which essentially makes them uncorrelated. Each *Principal Component* is basically a line that maximizes the variance of the features of the data. A line with higher variance means that more data points can be described by that line which means that it holds more information about the data. PCA maximizes the variance of each *Principal Component* by selecting the line that maximizes the average of the squared distances from any data point to the line. This line is the first *Principal Component*; PCA calculates the second *Principal Component* by doing the same process using the first *Principal Component* and its orthogonal line as axes. The same process happens until the total of *Principal Components* is equal to the number of the original features of the data.

It makes sense that not all *Principal Components* have the same variance, the first one has the greatest variance, and the second has the second greatest and so on.

In order to perform dimensionality reduction we can simply discard those *Principal Components* that have small variance and project our data in the space of the first few *Principal Components* which will allow us to keep most of the useful information of our data but with significantly smaller dimension. We are sacrificing a small amount of information in exchange for low dimension data that is easier to work with.

Dimensionality reduction is important to visualize high dimension data, detect latent features in the data, and reduce noise but most importantly to combat the Curse of Dimensionality and to increase the effectiveness of the Gaussian Mixture Model.

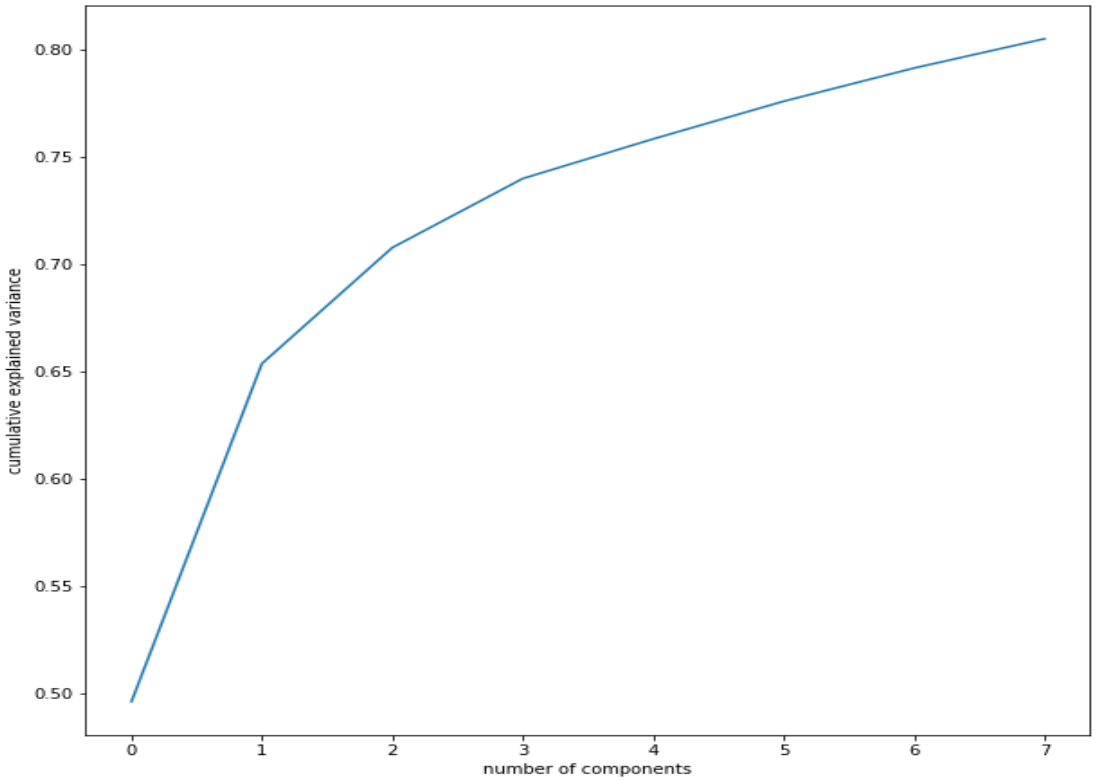


Figure 18: Variance ratio for each Principal Component.

We use $n = 8$ *Principal Components* to reduce the feature data that we extracted from the DeepLabV3+ neural network to a manageable dimension. This not only helps the fitting times to be drastically reduced but also the *Principal Components* contain superior semantic information.

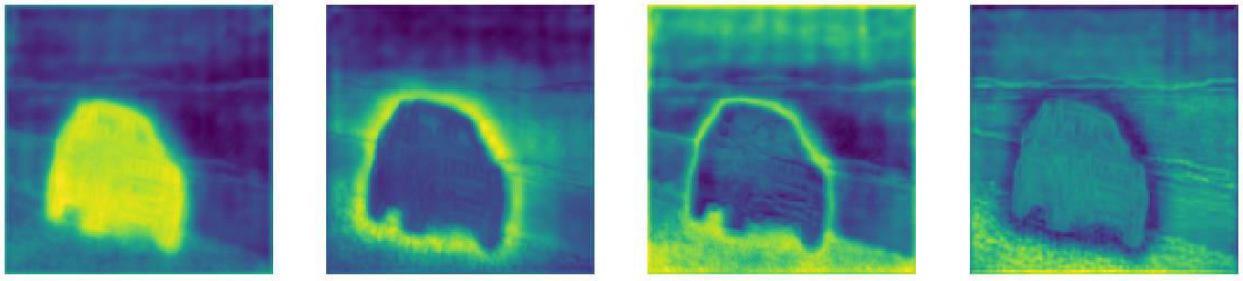


Figure 19: $n = 4$ *Principal Components.*

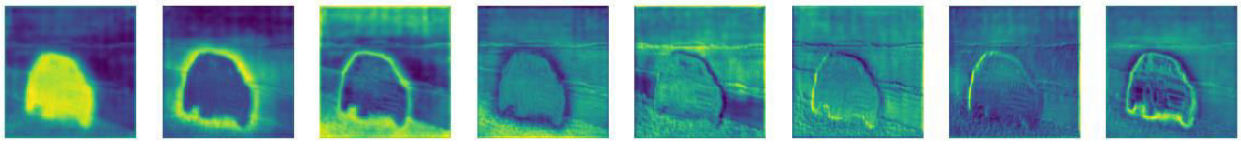


Figure 20: $n = 8$ *Principal Components.*

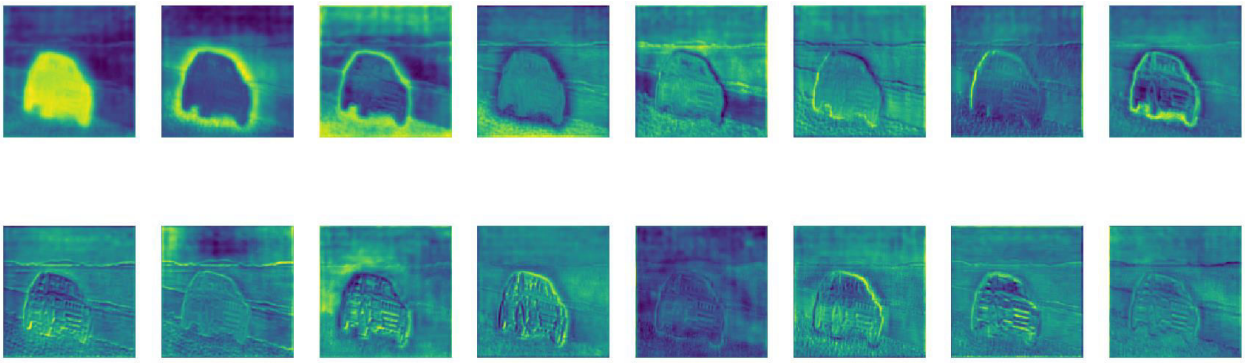


Figure 21: n = 16 *Principal Components.*

Filtering

Using simple image filters can go a long way in improving the result of the image segmentation. Filters can be used for a variety of image processing tasks such as denoising, sharpening and smoothing among others. In our case, the goal is to make the segmentation boarders of a segmented image clearer, as well as to remove noise created by the imperfections of segmentation.

Mean filter

One of the simplest linear filters is the mean filter. It is used for smoothing out the image and reducing the noise. The basis for this filter is replacing the value of each pixel with the mean pixel value in its neighborhood. This results in the reduction of variance locally in each pixel and therefore the blurring of the image. More specifically, for pixel (i, j) of an image I and neighborhood N the pixel value of that pixel is replaced using the following equation:

$$I'(i,j) = \frac{1}{M} \sum_{(x,y)\in N} I(x,y)$$

where M is the number of pixels in the neighborhood N. The neighborhood N is usually defined for each pixel since it corresponds to a square mask of size that is determined by the distance from the current pixel. Therefore, a distance of 1 defines a neighborhood of 3x3 which is fitted accordingly to the edges of the image. The bigger the neighborhood, the more intense the smoothing effect and the more details get blurred out. The mean filter can be considered a low pass filter that erases the higher pixel values which essentially correlate with details of the image. If we need to bring out the contribution of the pixels based on their distance from the center pixel, we can use smoothing masks like the following:

$$\frac{1}{16}\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix}$$

Median filter

The median filter is a linear low pass filter that is commonly used to reduce impulse noise on images. It finds the median pixel value in a set of pixels A. Starting with the set of pixels, it sorts the pixel values with ascending order which means that the median pixel value of the set is the center value of the sorting for an odd number of pixel values or the mean of the two neighboring center values for an even number of elements. More specifically for a set of pixels $\boldsymbol{A} = \{a_1, a_2, \dots, a_n\}$, sorted such that $a_1 \leq a_2 \leq \cdots \leq a_n \in R$, the median pixel value is defined as follows:

$$median(\boldsymbol{A}) = \begin{cases} a_{\frac{n+1}{2}} \; if \; n \; is \; odd \\ \frac{1}{2}\left(a_{\frac{n}{2}} + a_{\frac{n}{2}+1}\right) \; if \; n \; is \; even \end{cases}$$

The median filter has the desirable property of not affecting step functions as well as ramping functions. It does however affect pulse functions. The affects change according to the size of the neighborhood used for the filtering. The biggest drawback of the median filter with rectangular neighborhood is the destruction of thin lines and sharp corners. This problem can be mitigated with the use of different size of neighborhood. This filter has many different uses, for instance, the detection of the background on digital documents as well as the elimination of small lines on images. It can also be used with restrictions on the amplitude of the impulse noise related to the pixel values of a neighborhood. We can also adjust the size of the neighborhood based on the object of the image.

Dilation

Dilation is one of the two basic operators in the area of mathematical morphology, the other being erosion. It is typically applied to binary images, but there are versions that work on grayscale images. The basic effect of this operation on a binary image is to gradually enlarge the boundaries of regions of foreground pixels, usually white pixels. Thus areas of foreground pixels grow in size while holes within those regions become smaller.

The dilation operator takes two pieces of data as inputs. The first is the image which is to be dilated. The second is a usually small set of coordinate points known as a kernel. It is this kernel that determines the precise effect of the dilation on the input image.

The mathematical definition of dilation for binary images is as follows:

$$X \oplus K = \bigcup_{x \in X} K_x$$

where X is the set of Euclidean coordinates corresponding to the input binary image, and K is the set of coordinates for the structuring element.

Let K_x be the translation of K so that its origin is at x. Then the dilation of X by K is simply the set of all points x such that the intersection of K_x with X is non-empty.

The mathematical definition of grayscale dilation is identical except for the way in which the set of coordinates associated with the input image is derived. In addition, these coordinates are 3-dimensional rather than 2-dimensional.

To compute the dilation of a binary input image by this structuring element, we consider each of the background pixels in the input image in turn. For each background pixel we superimpose the structuring element on top of the input image so that the origin of the structuring element coincides with the input pixel position. If at least one pixel in the structuring element coincides with a foreground pixel in the image underneath, then the input pixel is set to the foreground value. If all the corresponding pixels in the image are background, however, the input pixel is left at the background value.

Erosion

Erosion is the second of the two basic operators in the area of mathematical morphology. The basic effect of the operator on a binary image is to erode away the boundaries of regions of foreground pixels usually white pixels. Thus areas of foreground pixels shrink in size, and holes within those areas become larger. Erosion is therefore the dual of dilation, which means that eroding foreground pixels is equivalent to dilating the background pixels.

The erosion operator, similar to dilation, takes two pieces of data as inputs, the input image and the kernel. The erosion operator takes two pieces of data as inputs. The first is the image which is to be eroded. The second is a usually small set of coordinate points known as a kernel. It is this kernel that determines the precise effect of the erosion on the input image.

The mathematical definition of erosion for binary images is as follows:

$$X \ominus K = \{x \in X | K_x \subseteq X\}$$

where X is the set of Euclidean coordinates corresponding to the input binary image, and K is the set of coordinates for the structuring element. Let K_x denote the translation of K so that its origin is at x. Then the erosion of X by K is simply the set of all points x such that K_x is a subset of X.

The mathematical definition for grayscale erosion is identical except in the way in which the set of coordinates associated with the input image is derived. In addition, these coordinates are 3-dimensional rather than 2-dimensional.

Opening

Opening and closing are two important operators from mathematical morphology. They are both derived from the fundamental operations of erosion and dilation. Like those operators they are normally applied to binary images, although there are also grayscale versions. The basic effect of an opening is somewhat like erosion in that it tends to remove some of the foreground (bright) pixels from the edges of regions of foreground pixels. However it is less destructive than erosion in general. As with other morphological operators, the exact operation is determined by a kernel. The effect of the operator is to preserve foreground regions that have a similar shape to this kernel, or that can completely contain the kernel, while eliminating all other regions of foreground pixels.

Opening is defined as an erosion followed by a dilation using the same kernel for both operations. The opening operator therefore requires two inputs, an image to be opened, and a kernel.

Closing

Closing is an important operator from the field of mathematical morphology. Like its dual operator opening, it can be derived from the fundamental operations of erosion and dilation. Like those operators it is normally applied to binary images, although there are grayscale versions. Closing is similar in some ways to dilation in that it tends to enlarge the boundaries of foreground (bright) regions in an image (and shrink background color holes in such regions), but it is less destructive of the original boundary shape. As with other morphological operators, the exact operation is determined by a structuring element. The effect of the operator is to preserve background regions that have a similar shape to this structuring element, or that can completely contain the structuring element, while eliminating all other regions of background pixels.

Closing is opening performed in reverse. It is defined simply as a dilation followed by an erosion using the same structuring element for both operations. See the sections on erosion and dilation for details of the individual steps. The closing operator therefore requires two inputs: an image to be closed and a structuring element.

Grayscale closing consists straightforwardly of a grayscale dilation followed by a grayscale erosion.

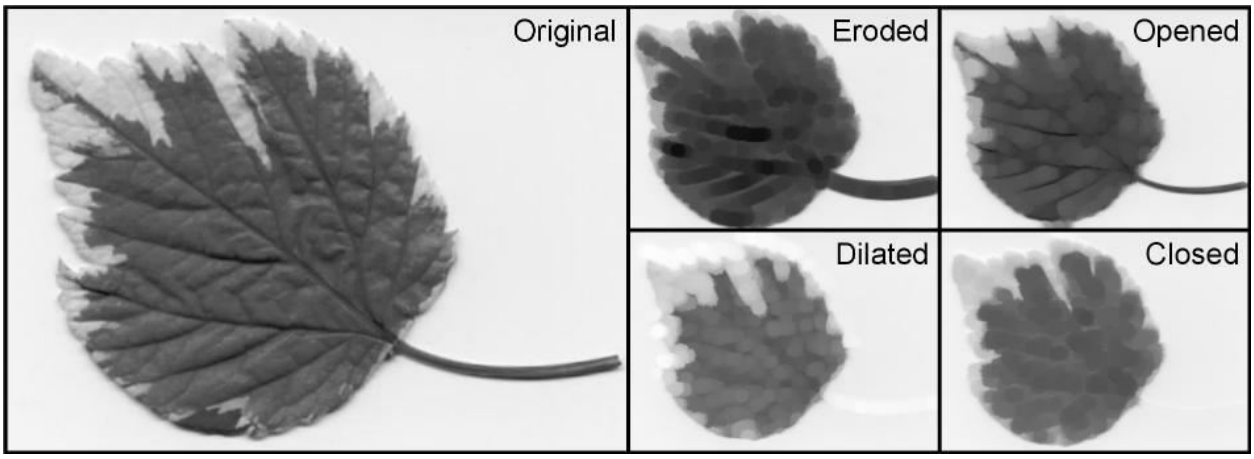
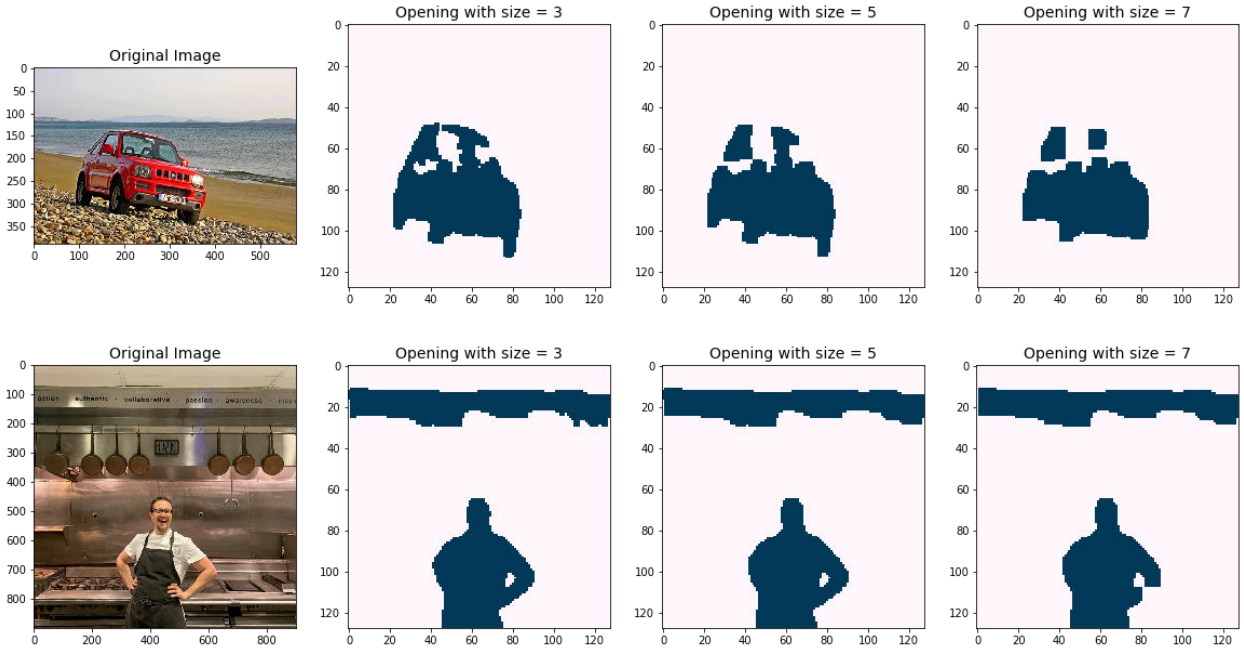


Figure 22: Different morphological filtering techniques.

In order to improve our results we decided to use a simple opening filter with a kernel of ones. Usually a kernel of size of 3x3 or 5x5 is used to remove noise from an image but the kernel size can change depending on how big of an effect we need the filter to have on the image. The same kind of process can be applied for the closing filter.



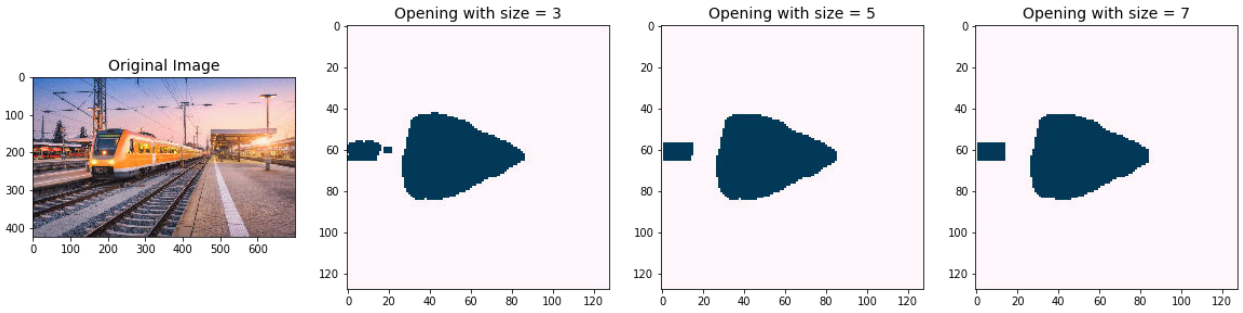


Figure 23: Opening filter using different kernel sizes.

Experiments

We initialized the DeepLabv3+ neural network using weights pretrained on the PASCAL VOC 2012 semantic segmentation dataset with the xception backbone for 21 classes. We used a collection of 8 images of objects that the DeepLabv3+ network can identify and we extracted the features that the network produces at the 405th layer for each of these images. Next we used PCA for n=8 Principal Components to reduce the dimension of the feature data. We derived with 8 Principal Component features for each image creating a tensor of 8x256x256x8, which we used to fit a Gaussian Mixture Model.

We started with a Gaussian Mixture Model with k=2 clusters. We proceeded by trying the results of the fitted Gaussian Mixture Model on 3 new images that were similar in terms of context to the initial 8 images. The way we processed the 3 new images is the exact same as the initial 8; first we used the DeepLabv3+ neural network to extract the features, then we performed Principal Component Analysis with n=8 components which gave us a tensor of 1x256x256x8 and finally we made a prediction on the resulting tensor using the Gaussian Mixture Model.

We also fitted a Gaussian Mixture Model for k=4 and k=8 clusters with the same 8 initial images to compare the results.

Figure 24: The original images.

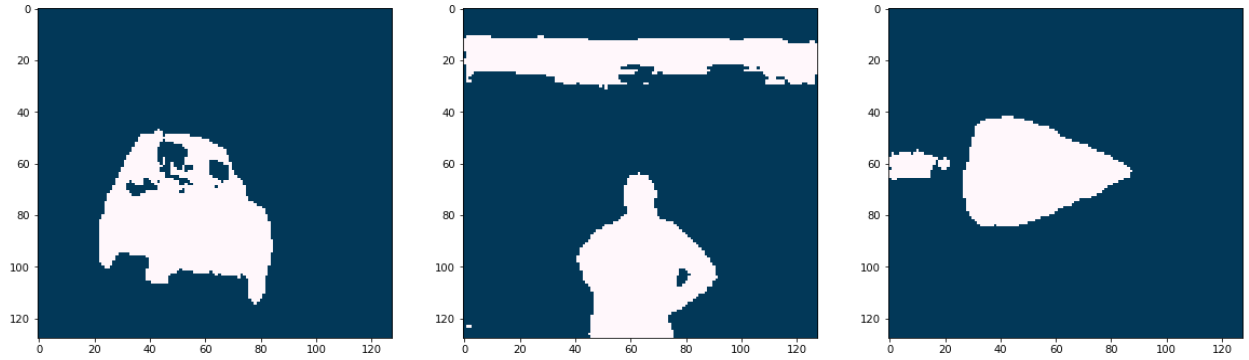


Figure 25: The segmentation result for a Gaussian Mixture Model of k=2 clusters.

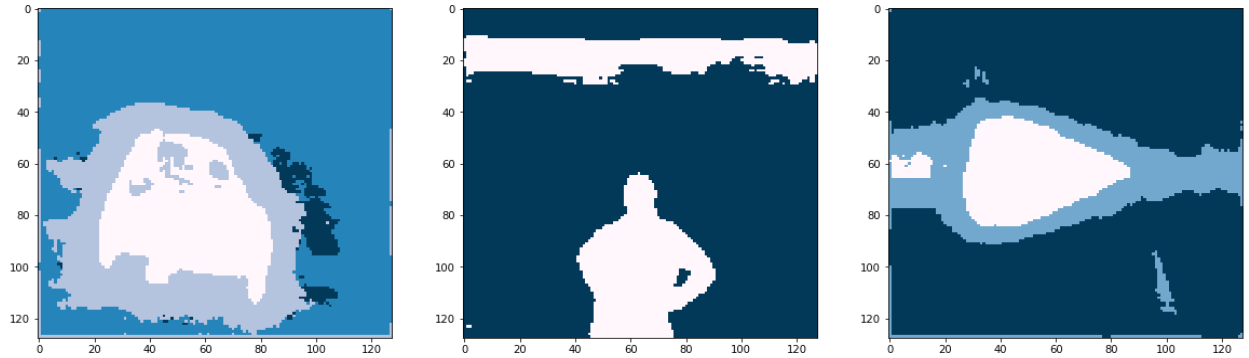


Figure 26: The segmentation result for a Gaussian Mixture Model of k=4 clusters.

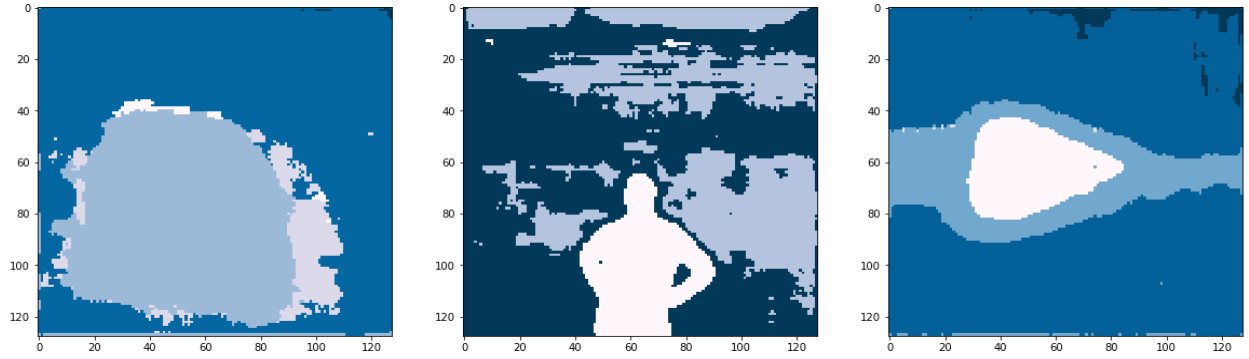


Figure 27: The segmentation result for a Gaussian Mixture Model of k=8 clusters.

Next we decided to use our technique on the frames of a video. We opted for a low resolution video of 640x360 pixels. For each frame in the video, we extract the features from that frame, perform Principal Component Analysis, then proceed to fit a Gaussian Mixture Model with the extracted features and finally make a prediction on the features. We used a Gaussian Mixture Model with k=2, k=4 and k=6 clusters and tracked the processing time for each frame. At this point we decided to also incorporate filtering to improve the results, so we used a max filter with window size of 5.

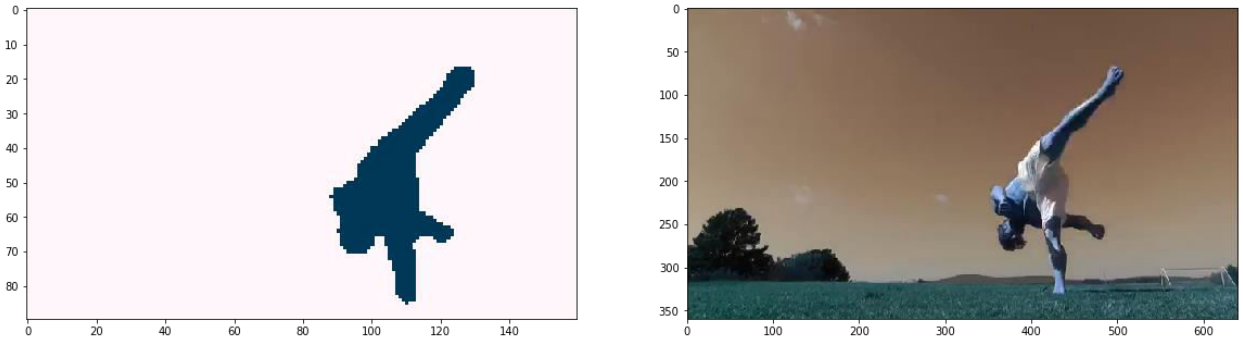


Figure 28: The segmentation result for a Gaussian Mixture Model with k=2 clusters, after filtering.

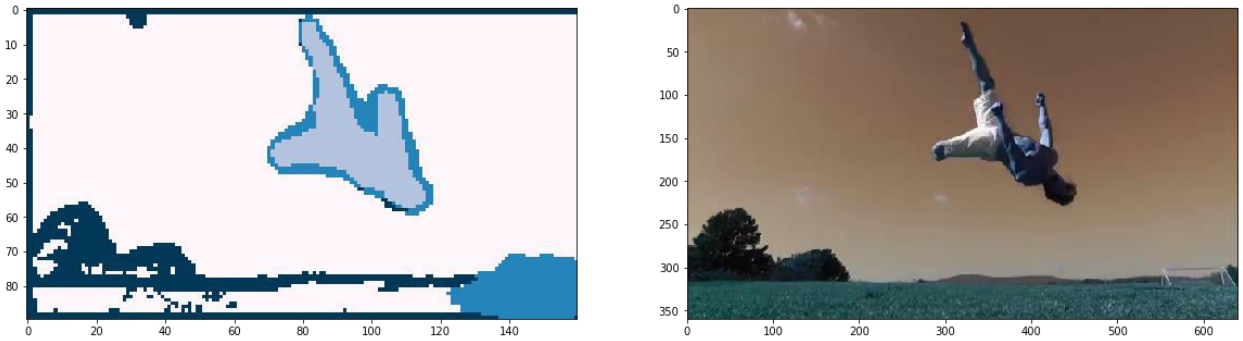


Figure 29: The segmentation result for a Gaussian Mixture Model with k=4 clusters, after filtering.

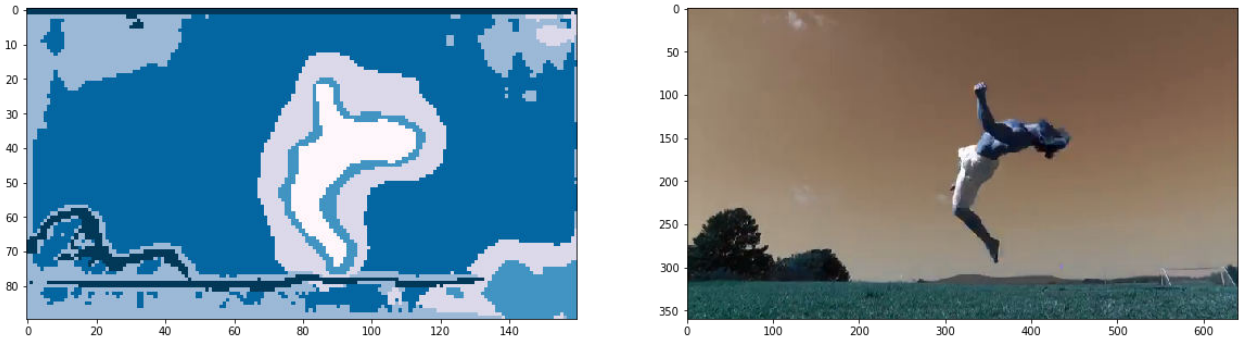
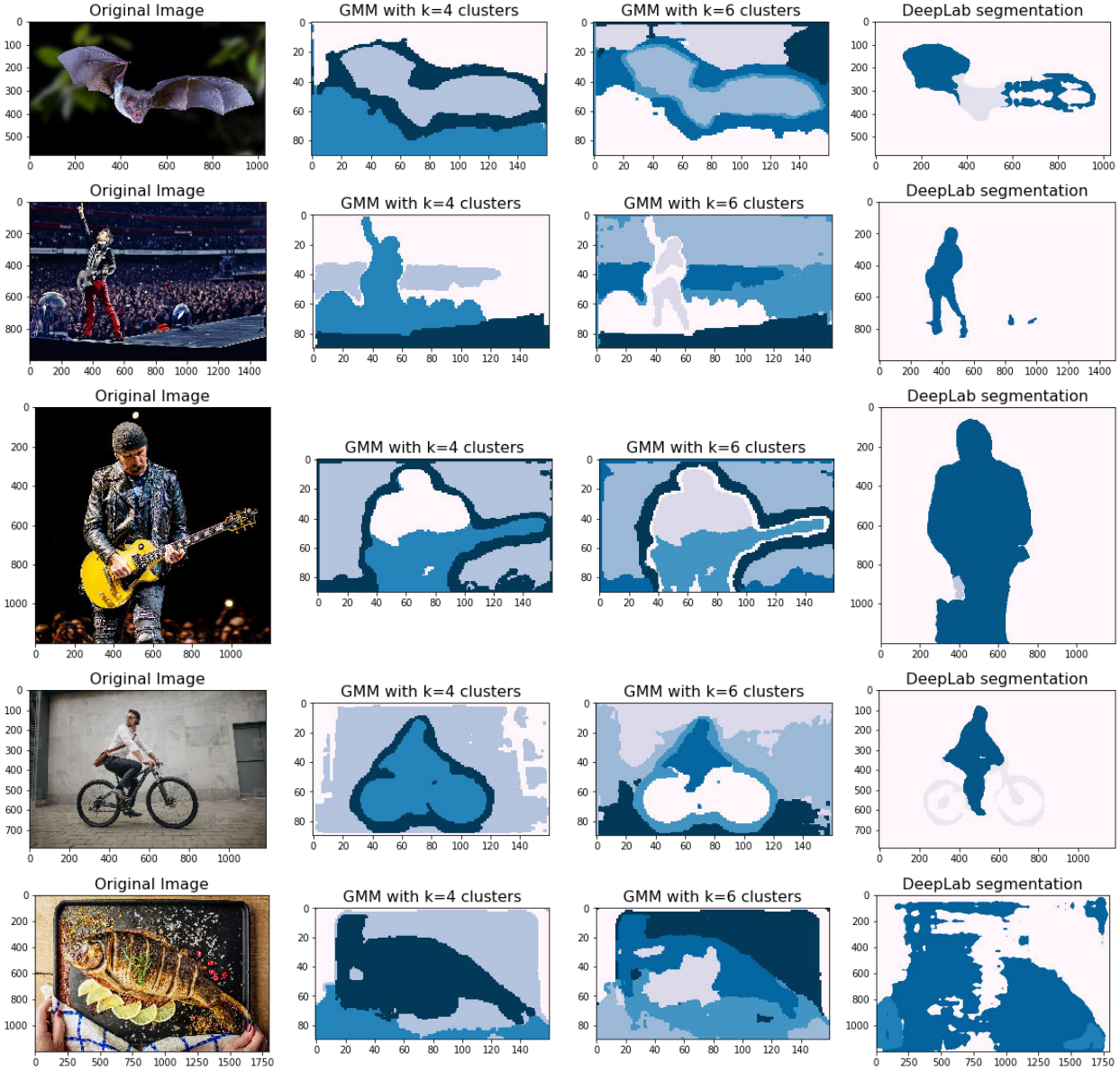


Figure 30: The segmentation result for a Gaussian Mixture Model with k=6 clusters, after filtering.

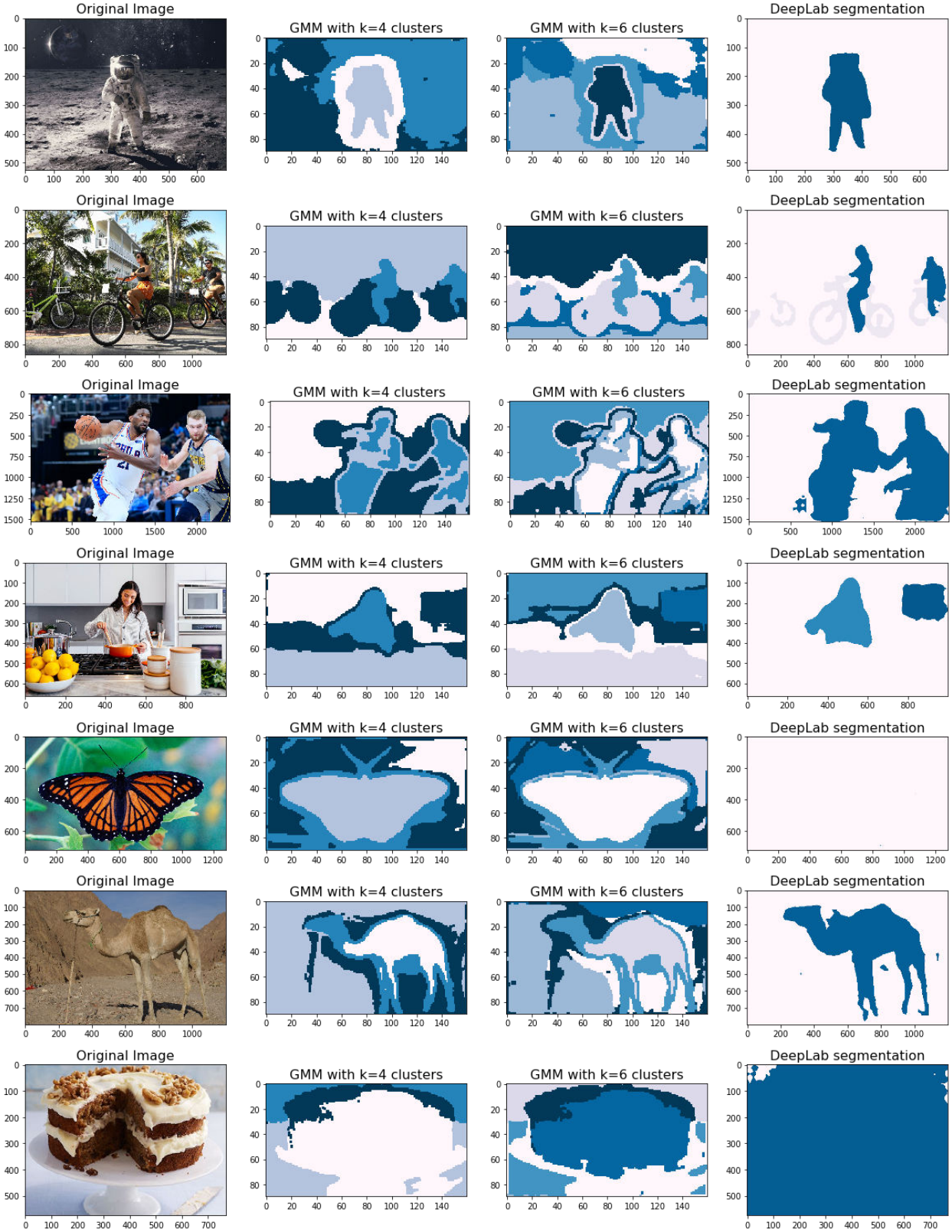
For the GMM with k=2 clusters the average processing time for a batch of 40 frames was 0.629 seconds or 1.59 frames per second. For the GMM with k=4 clusters the processing time was 0.632

seconds or 1.58 frames per second. For the GMM with k=6 cluster the processing time was 0.648 seconds or 1.54 frames per second.

In order to validate the results of our system, we compared the segmentation we were able to achieve for a small set of images with the original DeepLab neural network. We used a Gaussian Mixture Model with k=4 and k=6 clusters.




Original Image
GMM with k=4 clusters
GMM with k=6 clusters
DeepLab segmentation
Original Image
GMM with k=4 clusters
GMM with k=6 clusters
DeepLab segmentation
Original Image
GMM with k=4 clusters
GMM with k=6 clusters
DeepLab segmentation
Original Image
GMM with k=4 clusters
GMM with k=6 clusters
DeepLab segmentation
Original Image
GMM with k=4 clusters
GMM with k=6 clusters
DeepLab segmentation
Original Image
GMM with k=4 clusters
GMM with k=6 clusters
DeepLab segmentation
Original Image
GMM with k=4 clusters
GMM with k=6 clusters
DeepLab segmentation


Original Image
GMM with k=4 clusters
GMM with k=6 clusters
DeepLab segmentation
Original Image
GMM with k=4 clusters
GMM with k=6 clusters
DeepLab segmentation
Original Image
GMM with k=4 clusters
GMM with k=6 clusters
DeepLab segmentation
Original Image
GMM with k=4 clusters
GMM with k=6 clusters
DeepLab segmentation
Original Image
GMM with k=4 clusters
GMM with k=6 clusters
DeepLab segmentation
Original Image
GMM with k=4 clusters
GMM with k=6 clusters
DeepLab segmentation
Original Image
GMM with k=4 clusters
GMM with k=6 clusters
DeepLab segmentation


Original Image
GMM with k=4 clusters
GMM with k=6 clusters
DeepLab segmentation
Original Image
GMM with k=4 clusters
GMM with k=6 clusters
DeepLab segmentation
Original Image
GMM with k=4 clusters
GMM with k=6 clusters
DeepLab segmentation
Original Image
GMM with k=4 clusters
GMM with k=6 clusters
DeepLab segmentation
Original Image
GMM with k=4 clusters
GMM with k=6 clusters
DeepLab segmentation
Original Image
GMM with k=4 clusters
GMM with k=6 clusters
DeepLab segmentation
Original Image
GMM with k=4 clusters
GMM with k=6 clusters
DeepLab segmentation


Original Image
GMM with k=4 clusters
GMM with k=6 clusters
DeepLab segmentation
Original Image
GMM with k=4 clusters
GMM with k=6 clusters
DeepLab segmentation
Original Image
GMM with k=4 clusters
GMM with k=6 clusters
DeepLab segmentation
Original Image
GMM with k=4 clusters
GMM with k=6 clusters
DeepLab segmentation
Original Image
GMM with k=4 clusters
GMM with k=6 clusters
DeepLab segmentation
Original Image
GMM with k=4 clusters
GMM with k=6 clusters
DeepLab segmentation
Original Image
GMM with k=4 clusters
GMM with k=6 clusters
DeepLab segmentation

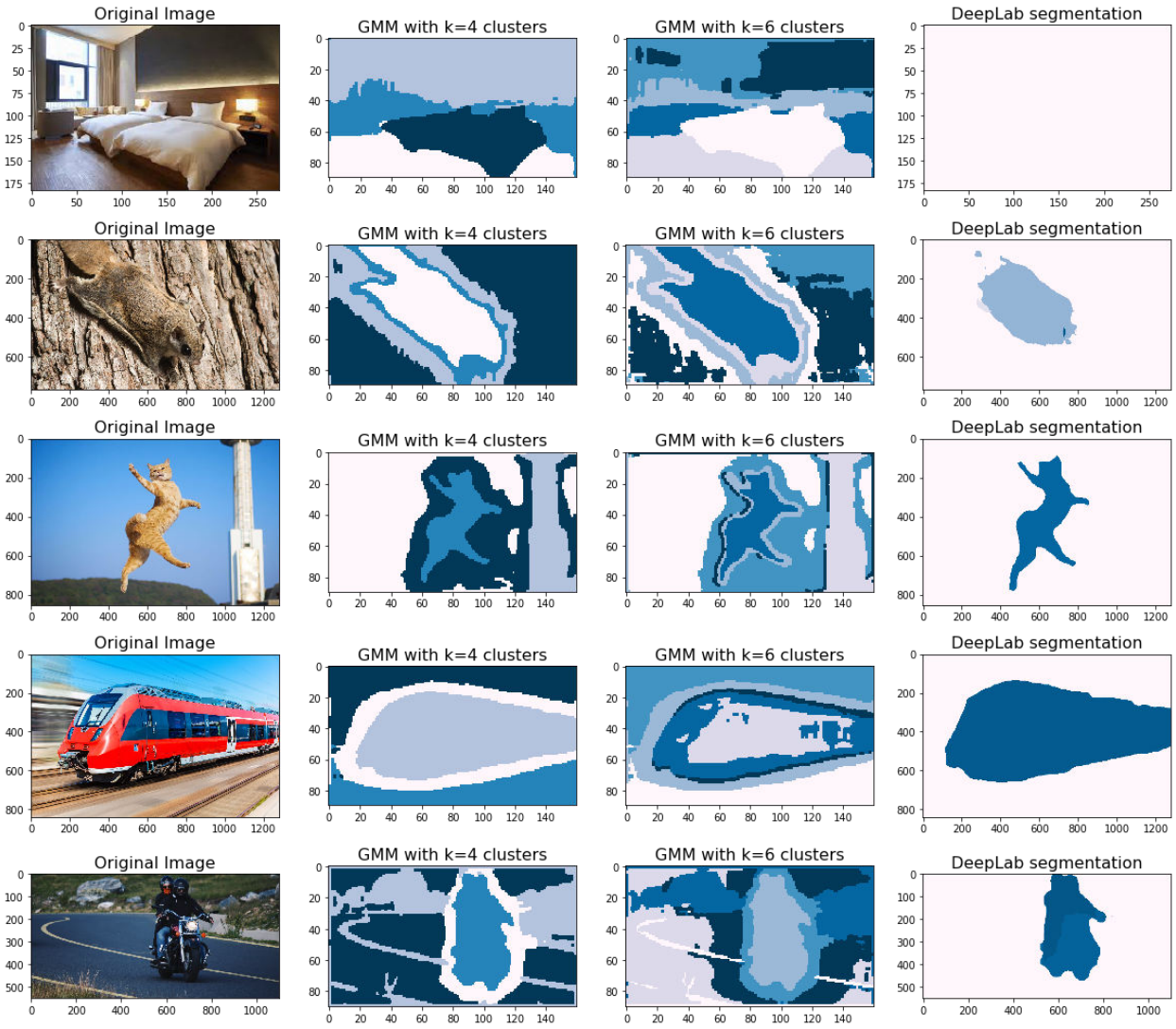


Figure 31: The results of our segmentation against DeepLab's segmentation.

Evaluating the results

A significant part of developing a computer vision system is establishing a way to quantify the quality of the results. In other words we need to measure how good the results are. To do this in our case we will use the *Intersection over Union* of the segmentation result.

This measure uses a manmade segmentation of the input image as ground truth and compares it to the segmentation from the system in question. To do this it uses the intersection of the segmentation result with the ground truth as well as their union. The intersection contains all the pixels that exist in both the ground truth and the resulting segmentation, and the union contains the pixels that exist in either one. Finally to calculate the Intersection over Union we divide the sum of

the intersection pixels with the sum of the union pixels. Intersection over Union is a ratio that can take values less than or at best equal to zero. The higher the Intersection over Union, the better is the segmentation.

Since our segmentation system uses more than one cluster, we need to discern the main cluster in order to compare it with the ground truth. This task can be exceedingly complex so we use a simple heuristic. The heuristic we use is based on the distance of each cluster from the center of the field of view of the segmentation result as proposed in [11]. This "dirty" heuristic is defined as follows:

$$\sum_{n=1}^{N_j} \|x_{jn} - c\|$$

where x_{jn} is the position of the n-th data point of the j-th cluster, and c is the position of the field of view center on the image.

This simple heuristic can wield good results for certain images but it may be inappropriate for other images. Using a more sophisticated heuristic could allow for better evaluation of the results, but in most of our test cases it is able to correctly detect the main cluster that corresponds to the ground truth. Some of our test cases are shown in the figures below:

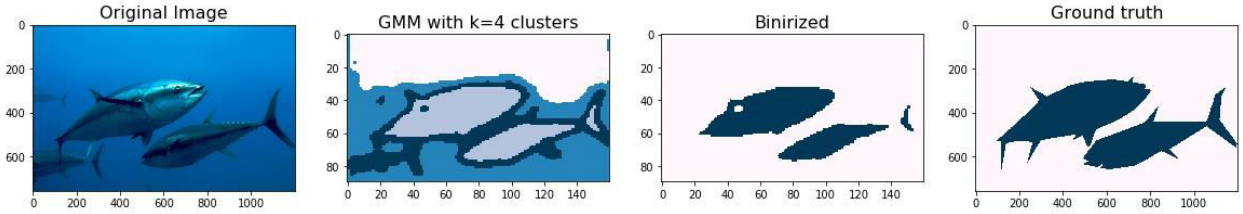


Figure 32: k=4 clusters with IoU of 61%.

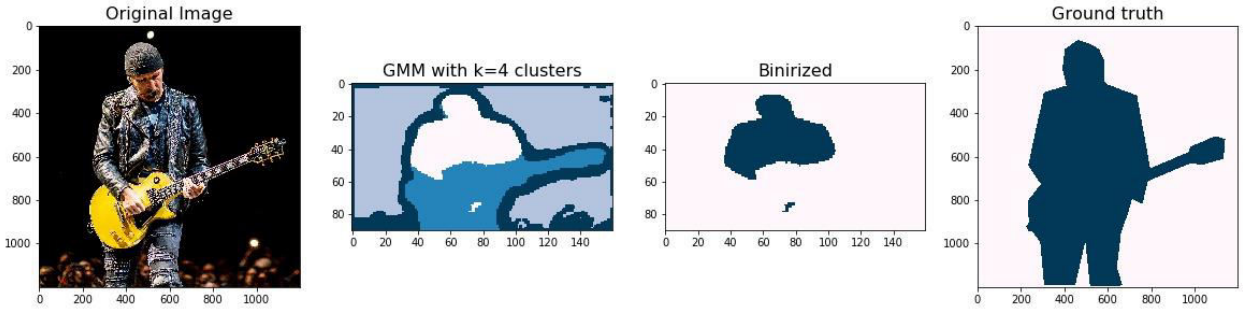


Figure 33: k=4 clusters with IoU of 41%.

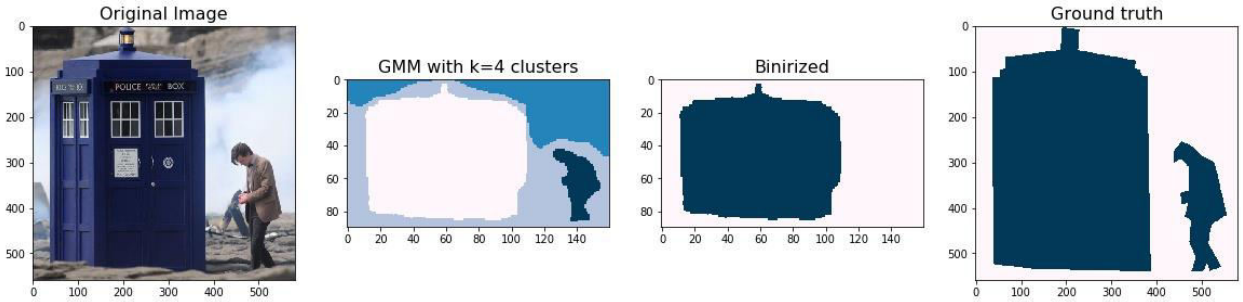


Figure 34: k=4 clusters with IoU of 80%.

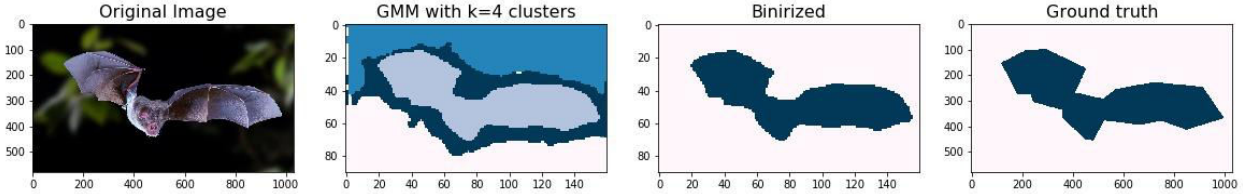


Figure 35: k=4 clusters with IoU of 86%.

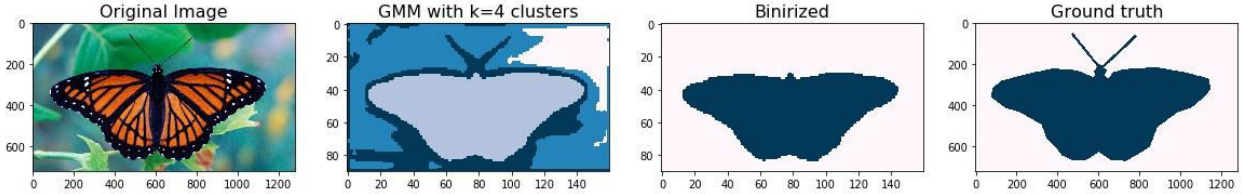


Figure 36: k=4 clusters with IoU of 85%.

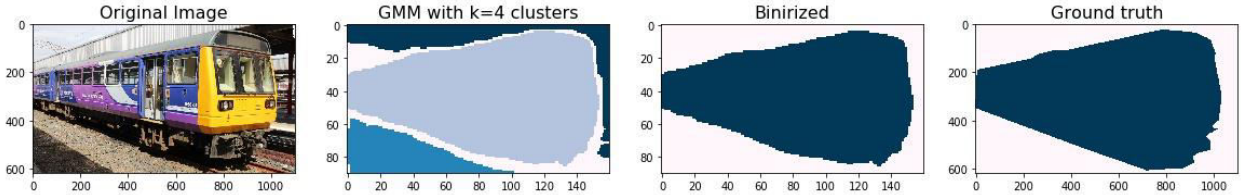


Figure 37: k=4 clusters with IoU of 90%.

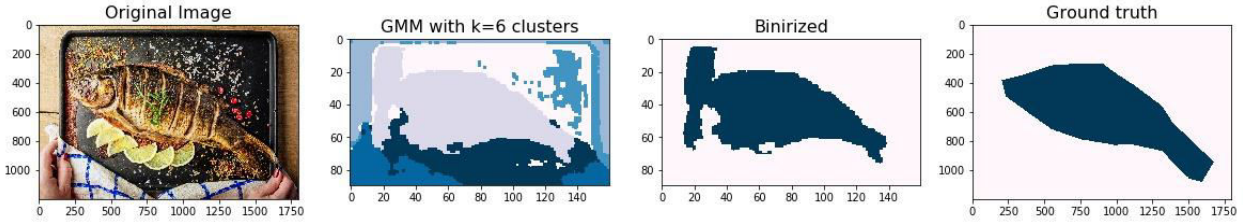


Figure 38: k=6 clusters with IoU of 68%.

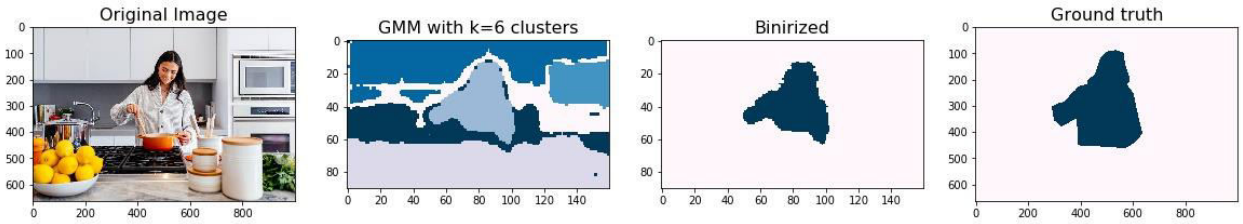


Figure 39: k=6 clusters with IoU of 70%.

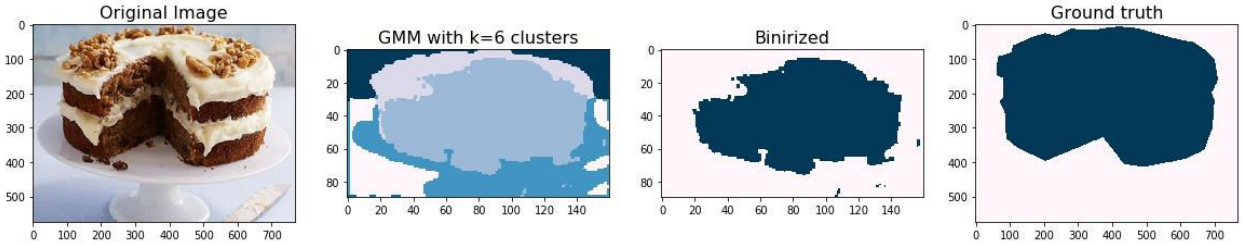


Figure 40: k=6 clusters with IoU of 64%.

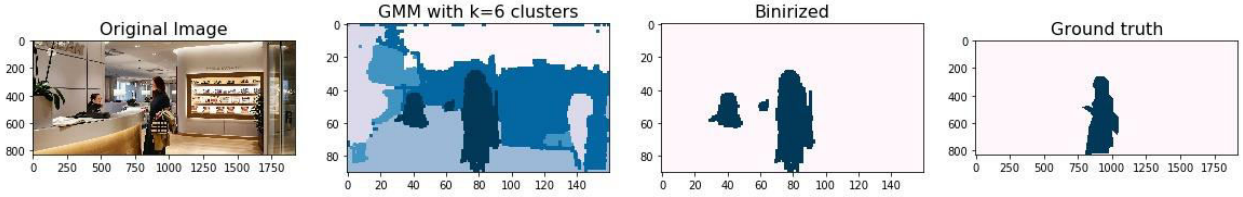


Figure 41: k=6 clusters with IoU of 51%.

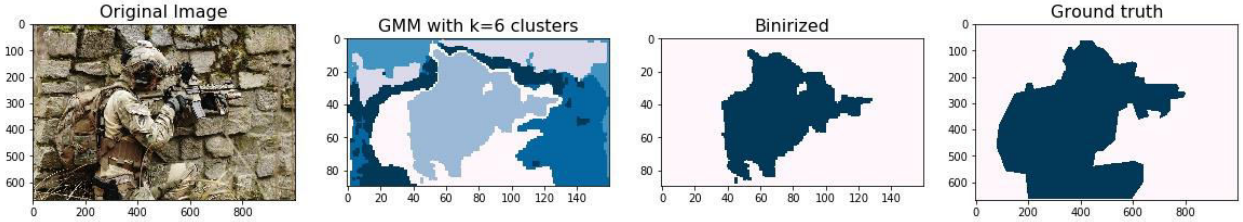


Figure 42: k=6 clusters with IoU of 54%.

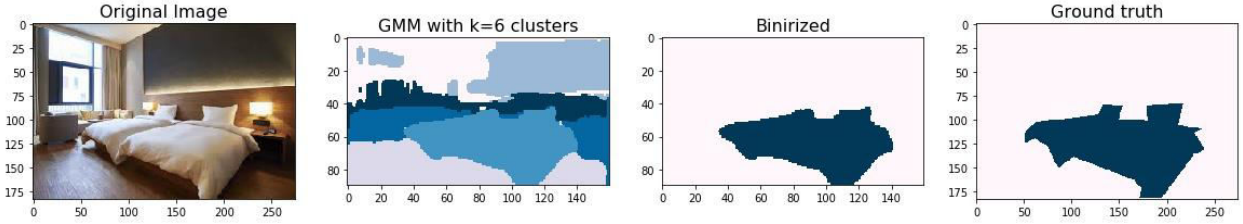


Figure 43: k=6 clusters with IoU of 80%.

Conclusion

Our proposed system offers a simple and easy to implement alternative to image segmentation, one that doesn't require a large dataset or labeled data. It does however require features of high quality, extracted from a state of the art Deep Convolutional Neural Network, a versatile clustering model and potentially some processing of the features.

The depth of the convolutional layer where the features are going to be extracted from needs to be sufficient for the features to capture useful patterns. In addition, we require a clustering method with the ability to model complex, real world data. The most important obstacle that needs to be overcome is the curse of dimensionality by performing dimensionality reduction on the deep features. Selecting the right number of clusters is an important decision, one however that depends on the task. Finally, post processing the results with filters can improve the segmentation.

One of the major advantages to this technique is that it doesn't require labeled data or big amounts of data to give descent results. This allows the system to generalize the semantic information it obtains from the deep features for a variety of tasks and images while it provides with fast inference despite the fact that Gaussian Mixture Model is not the fastest method for clustering.

The drawback of our system is that it requires features extracted from a state of the art neural network in order to achieve a good result. Without the deep features extracted from the Deeplabv3+ neural network, none of the results would have been possible. On top of that, it doesn't provide with any kind of classification of the objects in an input image.

A potential improvement is the post processing methods and the filters that we use. Using more advanced smoothing techniques such as a *Conditional Random Field* or a *Markov Random Field* could greatly improve the results. Using linear interpolation, we can adjust the size of the segmented image and resize it to our desired dimensions.

Another addition that could go a long way in improving the results is some feature engineering. Integrating the pixel positions into the features could allow the features to capture even more information about the input image.

References

[1] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L. Yuille. "DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs". In arXiv:1606.00915, 2017.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition". In arXiv:1512.03385, 2015.

[3] Karen Simonyan, Andrew Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition". In arXiv:1409.1556, 2015.

[4] Gao Huang, Zhuang Liu, Laurens van der Maaten, Kilian Q. Weinberger "Densely Connected Convolutional Networks". In arXiv:1608.06993, 2016.

[5] Md Zahangir Alom, Mahmudul Hasan, Chris Yakopcic, Tarek M. Taha, and Vijayan K. Asari, "Improved Inception-Residual Convolutional Neural Network for Object Recognition". In arXiv: 1712.09888, 2017.

[6] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, "Going Deeper with Convolutions". In CVPR, 2015.

[7] Kaiming He, Georgia Gkioxari, Piotr Dollár, Ross Girshick, "Mask R-CNN". In arXiv: 1703.06870, 2017.

[8] Olaf Ronneberger, Philipp Fischer, Thomas Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation". In arXiv: 1505.04597, 2015.

[9] Νικόλαος Παπαμάρκος. 2015. Ψηφιακή Επεξεργασία & Ανάλυση Εικόνας. ISBN: 978-960-92731-3-8

[10] Ian Goodfellow, Yoshua Bengio and Aaron Courville. 2016. Deep Learning. MIT Press. http://www.deeplearningbook.org

[11] G. Sfikas, Joao Patacas, Charalampos Psarros, A. Noula, D. Ioannidis, D. Tzovaras. 2019. A DEEP NEURAL NETWORK-BASED METHOD FOR THE DETECTION AND ACCURATE THERMOGRAPHY STATISTICS ESTIMATION OF AERIALLY SURVEYED STRUCTURES. http://cs.uoi.gr/~sfikas/convr-detection-and-thermal-determination.pdf