Object Classification Using Neural Network and Various Feature Extractors

ニューラルネットワークと特徴抽出器を用いた画像のクラス分類について

ZHAO, Xiaotian
趙　笑添
171600

March, 2017

Object Classification Using Neural Network and Various Feature Extractors

ニューラルネットワークと特徴抽出器を用いた画像のクラス分類について

付和文抄訳

A Thesis Presented to the Faculty of
the International Christian University
for the Baccalaureate Degree

国際基督教大学教授会提出学士論文

by

ZHAO, Xiaotian
趙　笑添
171600

March, 2017

Approved by _____
Professor
Kuticsne Matz, Andrea
Thesis Advisor
論文指導審査教授

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

We are living in a world with exponential increase of multimedia contents such as pictures and videos. Back to the days when we still use film cameras, we did not have a problem categorizes pictures based on what the picture is about. With the coming of the digital ages, there has been a massive increase of cameras all around the world such as camera built in cellphones, compact digital cameras, surveillance cameras. Handling such data by only us human, became no longer possible. Thus, we need the power of computer to automatically categorize and make sense to images or videos that we took instead of us. As of now, research such as [1, 2] are some of the front line at this specific task. However, even in these researches, simple task such as recognizing objects in a image as we human do is very challenging.

1.1 What is Object Classification?

Intuitively we know that the task of recognizing objects IS easy if all the objects are in the same shape and color. However, in reality, objects especially animals are very diversified. Even animals in a single category have different fur colors, sizes or shapes. They may also stretch themselves to create even more various shapes.

Even in these environment, human can still solve this problem easily. Say we want to look for a cat in a image, we know what cats are and differentiate them from dogs or rabbits. We can also recognize cats with different colors or in various poses. However,

for computer, this is a very difficult task.

We need to ask ourselves how we can characterize group of objects in extreme variations such as shape and appearance. One approach is to manually design a model for each object that we wish to recognize. Given an input, we find which group does the input belongs to by manually constructing feature extractors made for specific purpose and use support vector machine,a multiclass classifier, to classify them. The most highlighted method using this approach is Deformable Part-based Model [3], which models objects by constructing an abstract shape and separate into parts of features. However, since the models are created with manually designed feature extractors and the nature of support vector machine which can not handle great many features, they are not very goof at classify or recognizes natural images such as cats.

In recent years, the improvements of computer hardware such as Graphic Processing Unit (GPU), has highlight the classification with "Deep Neural Network" or "Deep Learning". Research such as [4] showed a big advantage of using Deep Learning in the field of object recognition and computer vision in general.

In deep learning, rather than extracting features with some particular filters, it uses huge volume of data to develop a system or filters that are automatically generated and can be used to extract features without our explicit engineering.

With such methods, Object Classification has improved dramatically. For example, CIFAR-10 datasets [5], which consists of 60,000 32x32 colour images in 10 different classes as shown on Figure 1.1, the classification accuracy of 80 %[6]in 2011 had improved to human level accuracy of 96.53% [7] in 2015.

However in a much more sophisticated classification challenges such as "Large Scale Visual Recognition Challenge (ILSVRC) Object Detection challenge which requires the participants to detect 200 classes of object inside an image such as one on Figure 1.2, have only reached 66 % at best in 2016 [8]. Though it is a huge improvements from

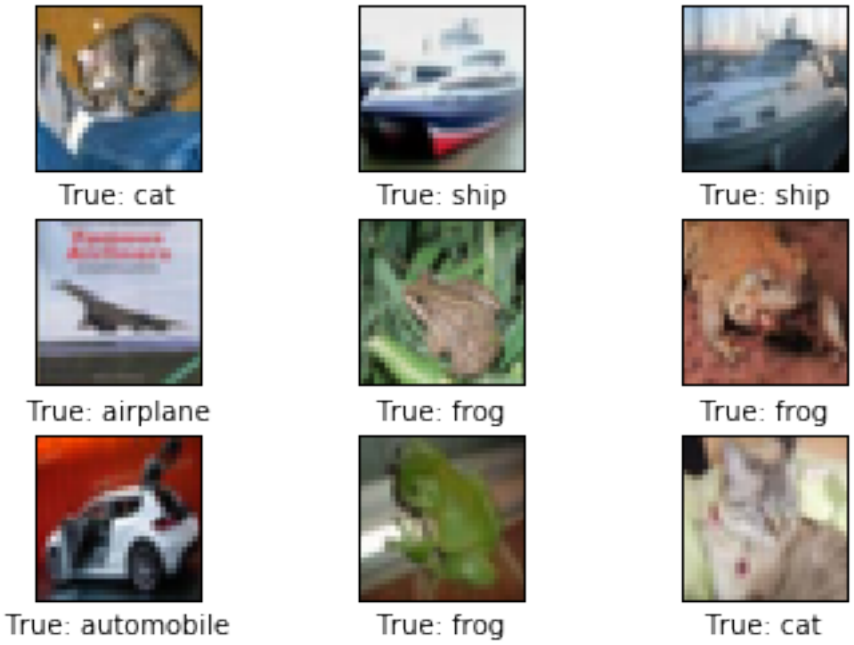


Figure 1.1: CIFAR10 Sample images

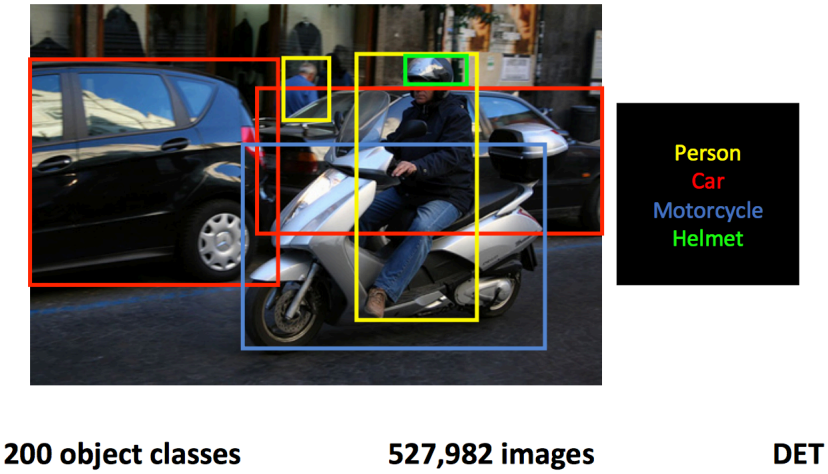


Figure 1.2: Object Detection Task on ILSVRC2016

23% in 2013, it still points out the fact that there is still a big room for improvements to train a network to conduct a task of listing objects in a picture.

1.2 Motivation

An object, in general, has their unique color, homogeneous texture and unique shape. These features of objects inside photos or videos have already being applied to describe image through the format of MPEG-7, one of the multimedia content description standard. It is being used to automatically label images based on their features which became the digital media library [9]. We believe the approach of extracting features from image as on MPEG-7 can be applied on object classification problems in neural network by strengthening their features in images before sending to the neural network as input images.

1.3 Related Works

This research is an attempt to combine two different approach on object classification, one uses manually constructing feature extractor such as Deformable Parted-based Model [3] and the other uses auto feature extractor such as neural network which is currently the most popular methods for the object classifications, detections tasks.

The anterior approach has already been used in application for a long time. For example, image descriptor such as MPEG-7 [9], a multimedia standard media descriptor that were used to retrieve similar images, are one of the major application.However, since the purpose of this standard is to comparing images, it has been designed to reduce data size and extract compact features from images or videos. This means that many features in an image that were not captured by that descriptor are diminished because it has the process of compressing. Thus we did not directly use the feature extractors of MPEG-7 as it is, but rather stimulating the similar operation or using their approach to find what features to extract.

The posterior approach, Neural Network, is still has not been much used as the anterior approach. It became highly recognized after the research by Y LeCun's leaded implementation of Convolutional Neural Network(CNN) which achieved 99%

accuracy with Handwritten digits classification task in late 90s [10]. Also another successful example using neural network was the implementation of multiple layers neural network (also known as Deep Neural Network) on "ImageNet Classification with Deep Convolutional Neural Networks" by Alex, Ilya and Geoffrey [4]. Their paper has brought the current trend of using neural network in object recognition and classification tasks. Since then, the accuracy of classification tasks has improved rapidly year after year [8]. Because of this highly accurate and capable classification model we can create with neural network, universities and companies started to invest and open source their own algorithm and libraries for even more development in the field of neural network. One of such library is "Tensorflow" [11] which is the open source version of Google's neural network tool used inside the company. Google had also brought their complicated pretrained network "Inception model" [12] along side with Tensorflow which enabled great many people with little computer resources to construct a practical classifier and predictor.

1.4 Our Approach

1.4.1 Objective

The goal of this paper is to find out whether feature extracted images with specifically designed extractors can improve the accuracy of classification taken by neural network. We chose 4 methods to extract features from images and implemented a convolutional neural network to analyze whether we can alter the behavior of the neural network to classify images focusing more on provided features and perform better at classification. We also tested our chosen feature extractor and use its output as the input to the trained network, such as Inception-Model by Google, to see if we can find the same effect on there as well.

We have tested with 3 different datasets and overall we successfully found that the effect of feature extracted images in the neural network and performed better at

classification when certain changes were added in the images. The detail discussion of results are written in chapter 5.

1.4.2 Outline of this paper

In Chapter 2, the general feature extraction approach of MPEG-7 and some other related feature extractors, that we believe can help on creating more accurate neural network, are explained.

In Chapter 3, the structure and mathematical concepts of neural network are briefly explained.

In Chapter 4, the process of implementing the feature extractors and neural network are presented.

In Chapter 5, we will discuss the result of this research.

In Chapter 6, we will conclude this paper and going through prospective future works that can be done based on this research.

The appendix contains some of the important source code that were used to conduct this research.

Chapter 2

Visual Descriptors

2.1 What can we know from a image?

A static image can be expressed in the following three category of descriptors in MPEG-7 [9].

- Color Descriptor

  The feature of color can be represented as distribution of the color, spacial layout and spacial structure.

- Texture Descriptor

  Texture is defined as the homogeneity or intensity found on color or monochrome image. This can be found at any fabric or even clouds, grass that found on natural environment.

- Shape Descriptor

  Shape can be explained as region and Contour found in an image. The most accurate extractor of such combines both Color and Texture of the images.

2.2 Color Layout Descriptor

The first feature we believe to have most of the information is color. Color Layout Descriptor is the most prominent example. The Color Layout Descriptor (CLD) is a compact, fast and resolution-invariant color representation method [13]. It is

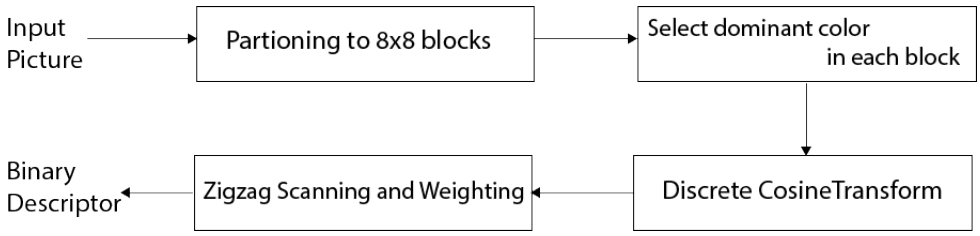


Figure 2.1: Diagram of Extracting CLD

capable of representing the spatial distribution of the colors. This is currently used in many similarity based retrieval, content filtering and visualization. For example, finding similar pictures or video frame from colored sketches. This extraction takes the following four stages on Figure 2.1

The input for this descriptor is a YCbCr color space, so all the images will require conversion from RGB color space to YCbCr. Then, these images are divided into uniform square 2D-array with width and height are 8. After this, at each block, a single color is calculated by taking the average color in that block. This 2D-array will then transformed by 8x8 Discrete Cosine Transformation(DCT) and returns 3 sets of 64 DCT coefficients. In the final step we take Entropy coding to compress the three matrix and quantize the low-frequency coefficient to avoid aliasing. In Chapter 4.2.2 we will go through the implementation of this descriptor.

2.3 Shape Extraction

Human can recognize objects by only the shape or boundary of an object [9]. This suggests the fact that shape are one of the biggest characteristics that we extract when we trying to understand an image. To find the shape of an object from object image can simply done by finding the outer closed region called contour. However, this becomes really difficult when there are region inside a region or some homogeneous texture which should be treated as one region. We need a method that will overcome these difficulties. We used two preexisting methods one is "Anisotropic Diffusion" [14] and the other is "Ultrametric Contour Maps" [15]. Both of these methods are capable

of extracting contour of objects. But, first we shall go through the basis of all these, the canny edge extractor which takes the pixel's brightness difference as edge.

2.3.1 Canny Edge Detector

Canny Edge Detector is the most highly optimized edge detection. It will not only takes the differential of pixels, but also take the orientation when deciding the edge lines.

1. Smoothing

   First, we remove noise using Gaussian smoothing. Gaussian which,defined as equation 2.1, are effective at removing salt-and-pepper noise. Single variable variance σ can be changed to apply different degree of the distribution.

   $$G(x) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2.1}$$

   Figure 2.2 (next page) shows the plot of Gaussian distribution with σ=1.4 and mean μ=(0,0) For the discrete representation of this Gaussian distribution, a 2D matrix called Gaussian kernel were used. equation 2.2 shows the Gaussian kernel $K(\sigma = 1.4)$.

   $$K(\sigma = 1.4) = \frac{1}{115} \begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix} \tag{2.2}$$

2. Taking Gradient

   In this process, following 2D filter (Gx and Gy) called Sobel filters are applied to the image.

   $$Gx = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \tag{2.3}$$

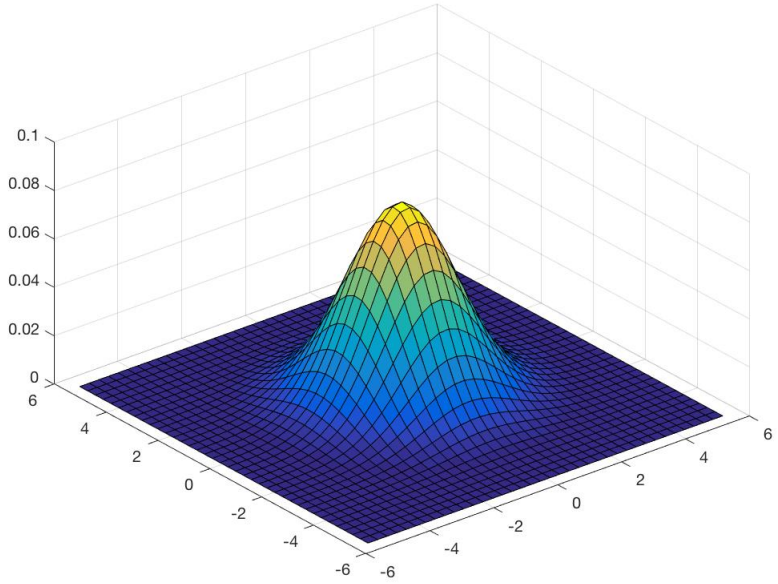


Figure 2.2: Gaussian distribution (μ, σ)=(0,1.4)

$$Gy = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{2.4}$$

Next, we evaluate the magnitude of the changes by taking the scalar of Gx and Gy.

$$G = |Gx| + |Gy| \tag{2.5}$$

3. Finding the edge direction

   We use arc tangent to find the direction of changes of gradient G

$$\theta = \arctan(Gy/Gx) \tag{2.6}$$

4. Grouping edge direction into 4 classes

   In image pixels we only have 4 types of slopes: 0, $\frac{\pi}{4}$, $\frac{\pi}{2}$, $\frac{3\pi}{4}$, we can group θ into 4 classes as shown on Figure 2.3.

   - IF $\frac{\pi}{8} < \theta < \frac{3\pi}{8}$ OR $-\frac{7\pi}{8} < \theta < -\frac{5\pi}{8}$: $\theta = \frac{\pi}{4}$
   - IF $\frac{3\pi}{8} < \theta < \frac{5\pi}{8}$ OR $-\frac{5\pi}{8} < \theta < -\frac{3\pi}{8}$: $\theta = \frac{\pi}{2}$

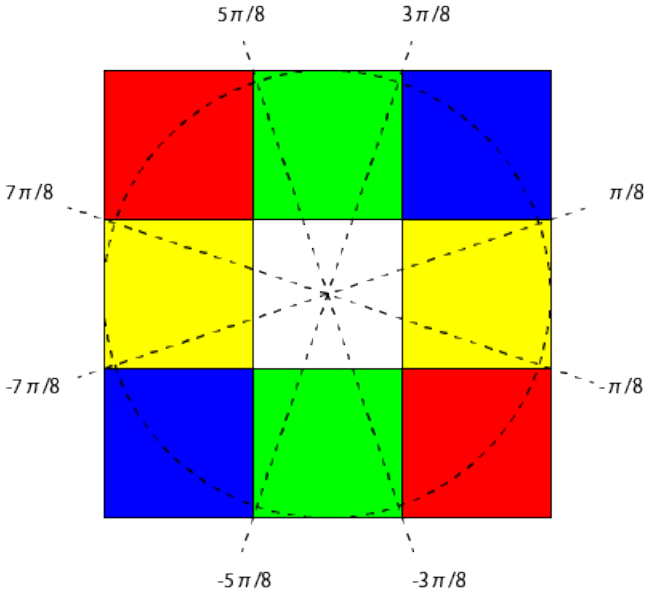


Figure 2.3: Grouping edge direction

- IF $\frac{5\pi}{8} < \theta < \frac{7\pi}{8}\ OR\ -\frac{3\pi}{8} < \theta < -\frac{\pi}{8} : \theta = \frac{3\pi}{4}$
- ELSE : $\theta = 0$

5. Apply non-maximum suppression

   In this process, we will thin lines constructed on the magnitude G defined in equation 2.5. Go through all the pixels and set them to 0 if they are not the largest pixels of its adjacent pixels.

6. Eliminating streaking with Hysteresis.

   We first set High threshold (HT) and low threshold (LT). Then the edge is defined when a pixel value at G is greater than HT, or between the two but adjacent to HT, it is considered to be Edge, otherwise not.

   - G > HT : Edge
   - HT> G > LT AND adjacent position's g is larger than HT : Edge
   - LT > G: NOT Edge

With the above process, edge are extracted from an image. Figure 2.4 shows the

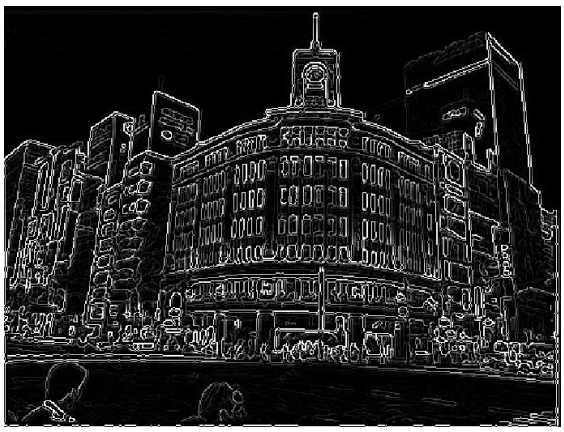
output from each process of the edge extraction. The implementation of this process are explained on chapter 4.2.3.

(a) Original Image

(b) Grayscale Image

(c) Smoothed Image

(d) After Taking Gradient



(e) After Nonmaximum Supression

(f) Canny Edge Image

Figure 2.4: Process of Canny Edge Extraction

2.3.2 Anisotropic Diffusion

"Anisotropic Diffusion" introduced by Perona & Malik [14] is used for removing noise in image but keeping important edges typically contour lines which are likely to represent the shape of the objects. Anisotropic Diffusion is developed upon "Isotropic Diffusion" which were inspired by heat equation. In this section we shall discuss both of these two.

Following [16] is the process of Anisotropic Diffusion

1. Isotropic Diffusion Diffusion is inspired by Heat Equation which are defined as equation 2.7.

$$\frac{\partial u}{\partial t} - \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) = 0 \tag{2.7}$$

   Here, $u(x, y, z, t)$ is the function of three spacial variables (x, y, z) and time variable t. In 2D images, we will consider function $I(x, y, t)$ where x, y are their Cartesian coordinate and t is it's time variable.

$$\frac{\partial I}{\partial t} = \alpha \left(\frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2} \right) \tag{2.8}$$

   This process is then simulated by convolve Gaussian kernel filter when implementing.

$$I_{i,j}^{t+1} = I_{i,j}^{t} + \lambda \left(I_{i-1,j}^{t} + I_{i+1,j}^{t} + I_{i,j-1}^{t} + I_{i,j+1}^{t} - 4I_{i,j}^{t} \right) \tag{2.9}$$

   However, this diffusion process are essentially same as Gaussian filtering which are designed to blur the whole image and edge lines or contour lines are not taken into account.

2. Anisotropic Diffusion In Anisotropic Diffusion, the diffusion equation used in Isotropic Diffusion are modified into equation 2.10 so that the diffusion process slows down or halts when it is approaching the boundaries of closed regions.

$$\frac{\partial I}{\partial t} = \operatorname{div} \left(c(x, y, t) \nabla I \right) = c(x, y, t) \Delta I + \nabla c \cdot \nabla I \tag{2.10}$$

here, $c(\|\nabla I\|)$ is a flux function that controls the rate of diffusion process. It will restrain the process of diffusion and enhances the edges when the boundaries of an object are approaching. The original paper proposed two definition of such $c(\|\nabla I\|)$ There are two functions below that can be used as flux function $c(\|\nabla I\|)$ in 2.11 or 2.12.

$$c(\|\nabla I\|) = e^{-(\|\nabla I\|/K)^2} \tag{2.11}$$

$$c(\|\nabla I\|) = \frac{1}{1 + \left(\frac{\|\nabla I\|}{K}\right)^2} \tag{2.12}$$

Free parameter K defines the strength of edge that should be counted as region boundaries. The larger this K value is, the more blurry it becomes.

When simulating with computer

$$I_{i,j}^{t+1} = I_{i,j}^{t} + \lambda \left[C_N \cdot \nabla_N I + C_S \cdot \nabla_S I + C_E \cdot \nabla_E I + C_W \cdot \nabla_W I\right]_{i,j}^{t} \tag{2.13}$$

N, S, E, W are up, down, left, right pixels from the target pixel (i,j).

The implementation of this method are shown on 4.2.4.

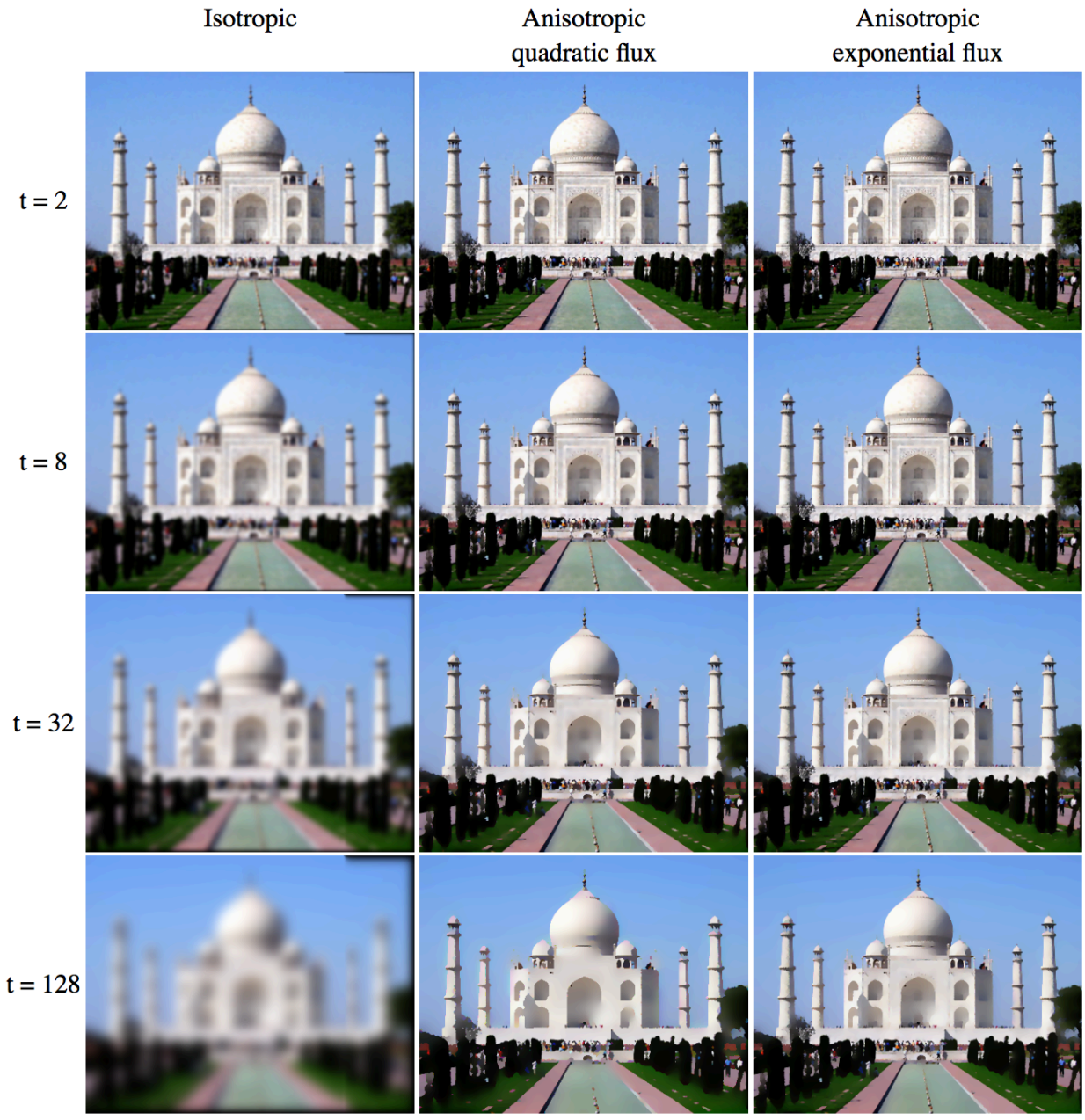


Figure 2.5: The left column shows Isotropic method, center column shows Anisotropic diffusion implemented with $c(\|\nabla I\|)$ with equation (2.12) and right column shows the one implemented with equation (2.11). t is the diffusion iterations.

Source:http://www.cs.utah.edu/~manasi/coursework/cs7960/p2/project2.html

2.4 Boundary Extraction in Natural Images Using Ultrametric Contour Maps

Ultrametric Contour Maps [15] is a method for extracting boundaries and segmentation tasks. It is calculated by the method called Region Merging which will construct

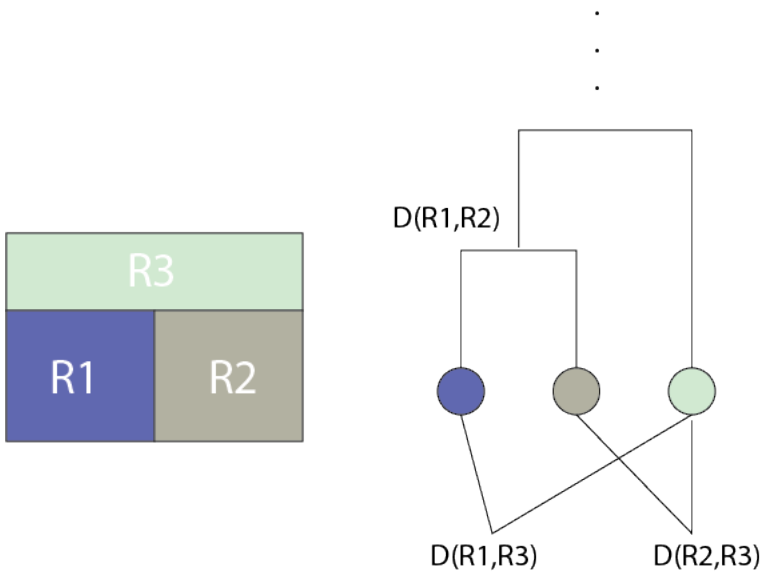


Figure 2.6: Hierachical Segmentaion methods

a nested partition of regions, given the distance of two region($R_1 and R_2$) $D(R_1, R_2)$ defined on equation 2.15, to construct an graph that are used in nested segmentation defined on 2.14. Figure 2.6 visualized this process.

$$D(x, y) <= max D(x, z), D(z, y) \tag{2.14}$$

- Mean boundary contrast $D_c(R_1, R_2)$, defined as max L*a*b difference within a given radius.
- Mean boundary gradient $D_g(R_1, R_2)$, defined as Pb(x)
- $D_a(R_1)$:Area + α Scatter in color space.

$$D(R_1, R_2) = [D_c(R_1, R_2) + a_1 D_g(R_1, R_2)] \cdot min D_a(R_1), D_a(R_2)^{\alpha_2} \tag{2.15}$$

By changing the parameter for the region merging process, we can make images that will have more contour or less. This is implemented in

Chapter 3

Deep Learning

3.1 Neural Network in Computer Vision

We can understand the hand written digit number. When we see a number, we can instantly tell what it is. Human have a primary visual Cortex consists of hundreds of millions of neurons dedicated to understanding images. In a sense, our brain can do things that is equivalent to what a super computer can do. As written in the introduction, There are two distinct approaches that is widely used to recognize objects from an image.

One of them is to construct a model that possess the characteristics of objects that we are trying to recognize. This techniques worked for static objects like mechanical parts, but natural objects such as plants, cats or even our handwritten numbers will not always shows clear features We find a shape that has circle in the bottom and a curve that connects to . However it becomes difficult if we try to explicitly make a program that can handle all kinds of '6' that human wrote. This becomes even more difficult when we try to recognize or classify cats, dogs and even human faces.

A different approach is taken in Deep learning. It uses large volume of data to develop a system that find the characteristic of the target objects. For instance, in the case of handwritten numbers classification, the famous dataset being used to train the network has 50,000 sample images **??**. Contrarily to the previous method, this can represent far more complicated model and moreover the extraction of such features

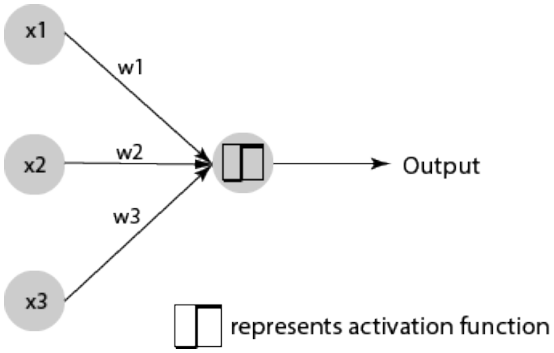


Figure 3.1: Perceptron with three input

are done automatically without us human to explicitly program it [17]. However at the same time, a practical system with this model requires a huge computer resources which was very difficult with computers in early days. In this chapter we shall go through some of the basic concept on neural network.

3.2 Feedforward Neural Network

3.2.1 Perceptrons

The Neural Network in our brain is the connection of neuron units. In Artificial Neural Network, this is simulated with Perceptrons [17]. As on Figure 3.1, perceptron takes several binary inputs, say $x_1, x_2, ..., x_n$, to produce one binary output. Then, weights $w_1, w_2, ..., w_n$ are multiplied to decide the importance of each input. The output of each perceptron are then evaluated by whether $\sum_j w_j x_j$ are less or greater than a threshold value.

$$output = \begin{cases} 0 & \sum_j w_j x_j \leq threshold \\ 1 & \sum_j w_j x_j > threshold \end{cases}$$

Now, define $b = -threshold$ and vectorize the multiplication of w and x, the notation above can be expressed simpler as follows.

$$output = \begin{cases} 0 & w \cdot x + b \leq 0 \\ 1 & w \cdot x + b > 0 \end{cases}$$

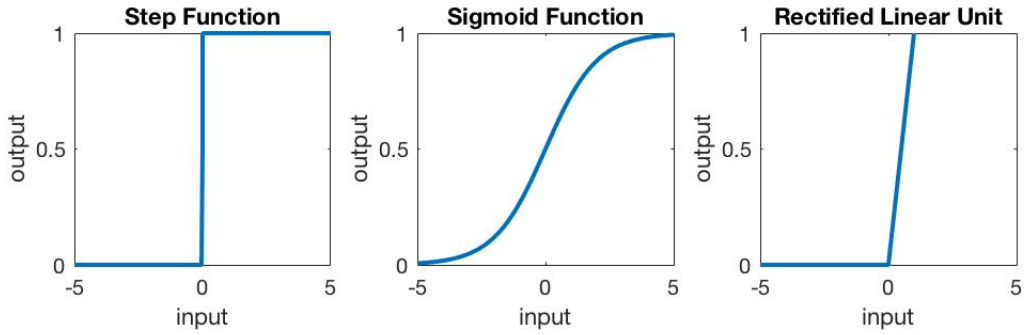


Figure 3.2: Graph of some Activation Functions

3.2.2 Activation Functions

We have so far discussed the output of each neuron being true or false binary which uses step function as activation function. However, it becomes a problem when we want to correct result by adjusting weights and biases when we tries to train the network because every little change on those values will immensely change the output of the perceptron [17]. To solve this issue, different activation function called "Sigmoid" is frequently used. Sigmoid, defined as (3.1), is a continuous function that varies from 0 to 1.

$$f(z) = \frac{1}{1 + e^{-z}} \tag{3.1}$$

z is the result of input multiplied with weights and adding bias ($z = w \cdot x + b$). Now we can express value like 0.187... and 0.675.... In recent years even simpler function such as "Rectified Linear Unit" (ReLU), formulated as equation 3.2, are becoming popular as the choice of the activation function for their less calculation and ability to abstract the input. Figure 3.2 shows the plots of function introduced above.

$$f(z) = max(0, x) \tag{3.2}$$

3.2.3 Structure of Neural Network

Combine multiple perceptrons we now have a network such as one on Figure 3.3. The input layer for every network is a flat vector, every high dimension data are reduced

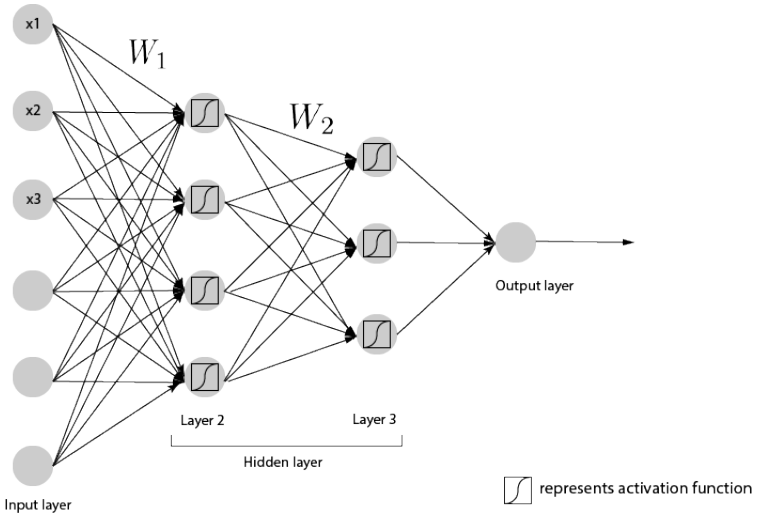


Figure 3.3: Example of fully connected neural network

to one dimension. The output layer is also a vector where it has as many perceptrons as many classes of objects. For binary classification problem, whether the output of the last neuron are over 0.5 or not decides whether the result is true or false. Between these two layers lies the hidden layers where main feature extraction and classification are conducted. By adding many hidden layers, we have network so called deep neural network or deep learning. Since we have more weight and bias parameters to tweak and express with this model, it is said that it performs better at classification.

3.2.4 Learning with Expected Output

We have so far learned the components of the neural network, now how do we learn or optimize network model that suits the expected output. This is done by shrinking the distance between outputs of the network and expected outputs [18].

First, we need a way to decide which class will a input image belongs. Let's say we have K classes to classify and we shall define $(u_k|u_k \in u, k \in K$ as $u_k = W \cdot z + b$ where z is the output from the previous layer with W, b being it's weight and bias at the very last layer. The output of this layer for class $(k|k \in K)$, y_k, are evaluated with Softmax function as on equation 3.3. This output y_k shows the probability for the input belongs to class k as visualized in Figure 3.4.

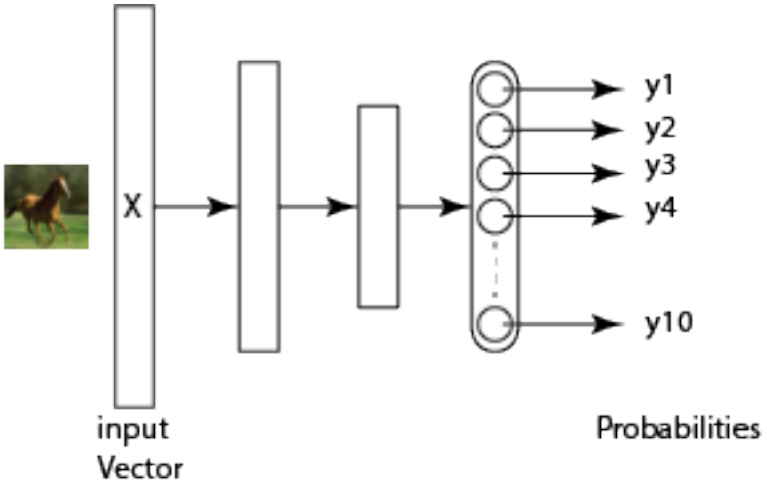


Figure 3.4: Example of multiclass classification problems

$$y_k = \frac{exp(u_k)}{\sum_{j=1}^{K} exp(u_j)} \tag{3.3}$$

Next, since y_k is the probability of input image x to be classified as class C_k,we can be express as on equation(3.4)

$$p(C_k|x) = y_k \tag{3.4}$$

Then, for mathematical convenience, we shall transform the expected output into "One-hot" encode representation which is a binary vector with it's value being true when it is the class of the expected output and false elsewhere. (3.5) shows the example of one-hot encoding d with class 4 being true class.

$$d = [0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0]^T \tag{3.5}$$

Given that there are N training data and K classes to classify,one-hot encoded vector of nth $(n|n \in N)$ expected output can be expressed as (3.6)

$$d_n = [d_{n1}\ d_{n2}\ d_{n3}\ ...\ d_{nK}]^T \tag{3.6}$$

Following this, the posterior distribution for n th input is

$$\begin{aligned} p(d_n|x_n) &= p(C_1|x)^{d_{n1}} p(C_2|x)^{d_{n2}} \dots p(C_k|x)^{d_{nk}} \dots p(C_K|x)^{d_{nK}} \\ &= \prod_{k=1}^{K} p(C_k|x)^{d_{nk}} \end{aligned} \tag{3.7}$$

Assume we will define the output of kth neuron (or class) in the very last softmax layer as $y_k(x; w)$, using the maximum likelihood estimation method, we can define the likelihood function $L(w)$ as the joint probability mass function. Say we have training data $(x_n, d_n)(n = 1, \dots, \ N)$ $L(w)$ can be expressed as equation 3.8.

$$\begin{aligned} L(w) &= \prod_{n=1}^{N} p(d_n|x_n; w) \\ &= \prod_{n=1}^{N} \prod_{k=1}^{K} p(C_k|x_n)^{d_{nk}} \\ &= \prod_{n=1}^{N} \prod_{k=1}^{K} (y_k(x; w))^{d_{nk}} \end{aligned} \tag{3.8}$$

However, high dimension functions are difficult to use, so we take the log of equation 3.8 and negate it to transform a more easier function called error function E(w) on equation 3.9.

$$E(w) = -\sum_{n=1}^{N} \sum_{k=1}^{K} d_{nk} \log y_k(x_n; w) \tag{3.9}$$

The error function above is also called "Cross Entropy" and the goal is to reduce $E(w)$ through many steps of learning.

3.2.5 Reducing Error

To reduce $E(w)$, a method called "Gradient Descent" are used repetitively to update $W(w_1, w_2 ... w_M)$. The rate of this update can be changed with different constant learning rate value ϵ as on equation 3.11.

$$\nabla E = \frac{\partial E}{\partial W} = \left[\frac{\partial E}{\partial w_1} \ \frac{\partial E}{\partial w_2} ... \frac{\partial E}{\partial w_M} \right]^T \tag{3.10}$$

$$W^{t+1} = W^t - \epsilon \nabla E \tag{3.11}$$

Algorithm 1: Backpropagation Algorithm

1. Define training sample as$z^{(1)}$($z^{(1)} = X_n$), calculate the layer output $u^{(l)}$ and network output $z^{(l)}$ sequentially (forward pass)
2. Evaluate $\delta_{ji}^{(L)}$ at the output layer as $\delta_{ji}^{(L)} = z_j - d_j$
3. Calculate $\delta_{ji}^{(l)}$ for all middle layer l=(L-1,L-2,...,2) as equation () from backward (backpropagation)

$$\delta_j^l = \sum_k \delta_k^{(l+t)}(w_{kj}^{(l+1)} f'(u_j^{(l)}))$$

4. Calculate $w_{ji}^{(l)}$ for the layer l=(2,3,...,L)

$$\partial E_n \partial w_{ji}^l = \delta_j^{(l)} z_i^{l-1}$$

Data: Training sample X_n, Expected output d_n
Result: Partial derivative of error function $E_n(w)$ for each layer l's weights.$\frac{\partial E_n}{\partial w_{ji}^l}$

However, in practice, we will not use all the training data all at once, but randomly selected partial data called "Batch". Assume we have N training samples, $E(W)$ are expressed as (3.12).

$$E(W) = \sum_{n=1}^{N} E_n(W) \tag{3.12}$$

Update (3.11) as (3.13) where every time W is updated, another E_n based on new sample are given.

$$W^{t+1} = W^t - \epsilon \nabla E_n \tag{3.13}$$

3.2.6 Backpropagation

In order to update the weights, ∇E_n needs to be calculated for every connection between the neurons. This is very computationally intense and had become a big deadlock in the development in the neural network. Backpropagation techniques ease this computation by utilizing the chain rule that is used in determining $\frac{\partial E_n}{\partial w_{ji}^l}$ at deep layer. The algorithm for this is explained on **Algorithm 1**.

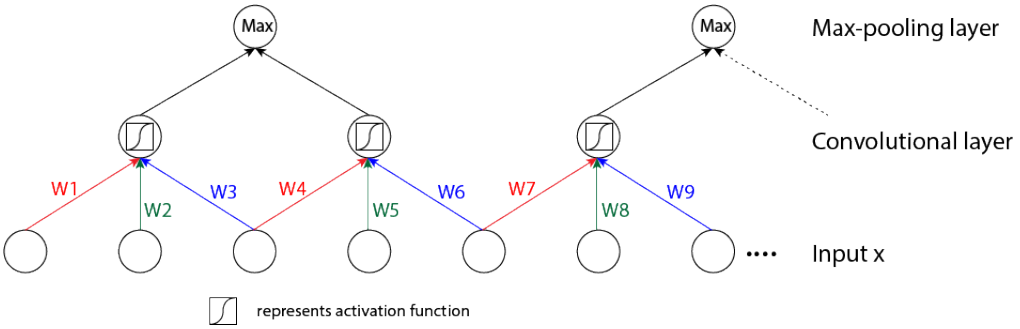


Figure 3.5: Convolution Explanation

3.3 Convolutional Neural Network

The network that we discussed so far are so called fully connected network where all the neurons on one layer connects to each neuron on the next. However this does not work well it is 2D image data. This is because neural network takes vector as input and spacial correlation of the original data becomes very weak. To overcome this problem, we can force the neurons to take small number of local connections. For example, we can pass adjacent pixels as input to each neurons. In this way, we can obtain a deep locally connected network similar to the neurons in our brain which are also mostly locally connected. We can still reduce amount of computation using the method called "weight sharing". In weight sharing, parameters that have same spacial position of the local pixels will have the same weight values. For example, in the figure 3.5, weights arrow with same color will have same weights: W1=W4=W7, W2=W5=W8, W3=W6=W9.

In signal Processing the operation above is called convolution where we apply a "filter" that contains the set of weights, to partial position of a signal. In 2D, it is a 2D matrix of weights, normally we use 3x3 in order to keep high locality. Let the image x to be $W \times W$ pixels image indexed by (i,j), and convolution filter h to be $H \times H$ pixels with (p,q) as index. The convolution is defined as equation 3.14.

$$u_{ij} = \sum_{p=0}^{H-1} \sum_{q=0}^{H-1} x_{i+p,j+q} h_{pq} \tag{3.14}$$

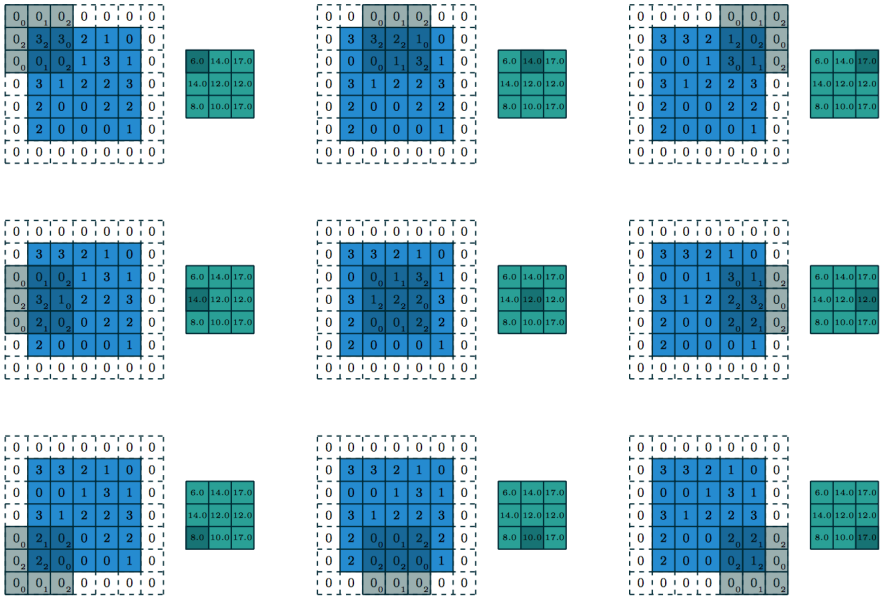


Figure 3.6: Calculation of Convolution procedure

Figure 3.6 shows the convolution calculation with W_1 as a filter with strides(number of pixel to shift between each convolution operation) = 2 and padding(number of zeros concatenated at all sides of matrix) = 1

$$W_1 = \begin{bmatrix} 0 & 1 & 2 \\ 2 & 2 & 0 \\ 0 & 1 & 2 \end{bmatrix} \tag{3.15}$$

Note that the filter W_1 is a set of variables we will be optimizing through the process of learning.

When the image is colored (multiple channels), or calculating convolution after the first layer, each channel are given its own weight kernel and perform the convolution operation as described above. After this, the output are added together to create a matrix. Using this matrix as input, the activation function are used to calculate the output of the convolution layer called "Feature Map". The number of feature defines how many this operation are going to take in the network, so if we have 10 features in a convolution layer, there will be 10 separate convolution operation and 10 feature maps. Figure 3.7 visualize this procedure. We shall go back to this method in section

4.3.2 and implement this with a neural network library.

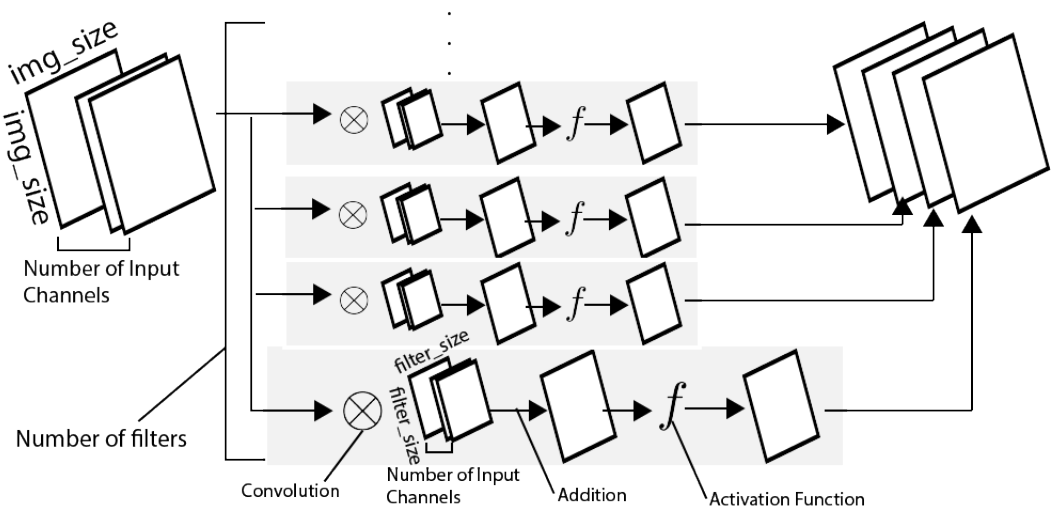


Figure 3.7: Convolution with multiple channels and filters

3.4 Pretrain with Autoencoder

In a basic neural network, starting weights and biases are randomly generated. However, we could tell the network to just go through the image to get the abstract of the training images that they will be trained with. This is what happened inside what is being called "Autoencoder".

Autoencoder is an unsupervised learning algorithm which uses back-propagation to set the output as close as input values. A process called "encode"(equation 3.4) and its reverse process "decode" (equation 3.5) is being applied on input.

$$y = f(Wx + b) \tag{3.16}$$

$$\hat{x} = \tilde{f}(\tilde{W}y + \tilde{b}) \tag{3.17}$$

From the two equations above the autoencoder can be explained as

$$\hat{x} = \tilde{f}(\tilde{W}f(Wx + b) + \tilde{b}) \tag{3.18}$$

After this, the optimization of $W, \tilde{W}$ and $b, \tilde{b}$ are conducted same as on learning process explained on (3.2.4). To apply the result to main network, the same network

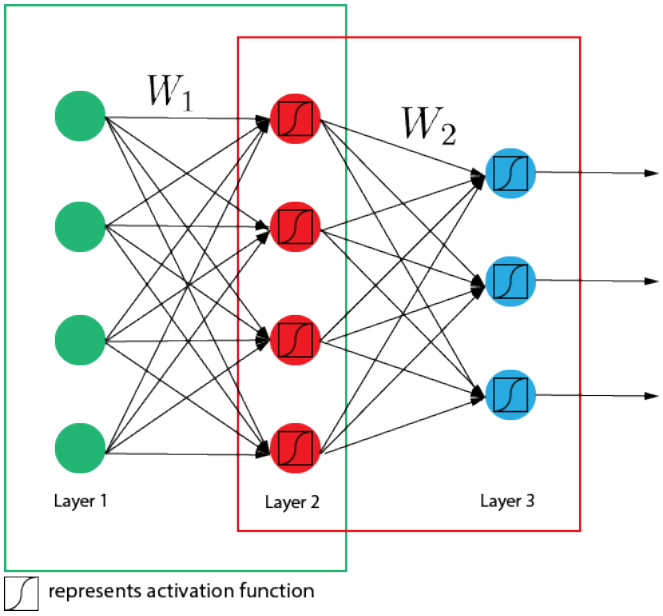


Figure 3.8: Dividing layers to pre-train

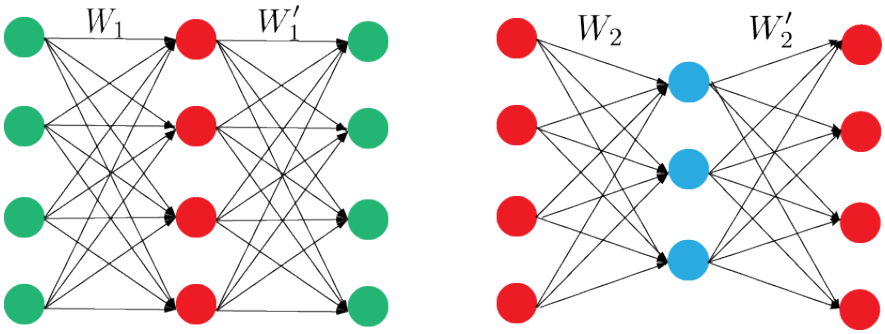


Figure 3.9: Training layer individually

will be constructed on Autoencoder and apply the above learning process through all consecutive layer one after another. Figure 3.8 and 3.9 shows this process. Weights and biases that are optimized by Autoencoder are then becomes the initial values when constructing the main network [19]. The implementation for this network can be found at section 4.3.3.

Chapter 4

Our Implementation

In our research, TensorFlow were used as the main Neural Network Construction and Matlab were used mainly for image processing. we shall go through these two tools in detail.

4.1 Tested Machine Specification

Table 4.1 shows the specification of our machine used in this research. The memory 12GB is very small for conducting this kind of research, so we recommend using more larger memory such as 16 GB or 32GB.

4.2 Image Transformations

We had conducted all feature extraction methods in chapter 2 using online open-source and language MATLAB. We shall go through its' process one by one.

Table 4.1: Detail specification of the testing machine

CPU	Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
Memory	12GB
GPU	GeForce GTX 1060 ARGMOR 6G OC by msi
Storage	120GB SSD + 500 GB HDD
Software	Matlab 2016a, CUDA tool-box 8.0, Tensorflow r0.12
OS	Linux Ubuntu 16.10

4.2.1 MATLAB

MATLAB [20] is a proprietary programming language that is developed by MathWorks that features High-level language specialized in scientific and engineering computation. It also offers variety of toolboxes such as Image Processing Toolbox, Statistics, Machine Learning Toolbox and other toolboxes that makes our field of study much easier. Moreover, many research on Image Processing and Computer Vision are also conducted with language MATLAB, which made it easy to try latest works by other researchers.

4.2.2 Color Layout Descriptor

In Color Descriptor, the key is to extract an abstract image. Since we know that neural networks would construct abstract image in the process of learning. However, in neural network, there were no operation that does discrete cosine transformation nor zig-zag scanning to reduce the size of images, we decided to take the operation of partitioning image to 8x8 and fill each block with their dominant color. We first tested with monochrome handwritten digits images and tested the effective of this operation on neural network. Figure 4.1 and Figure 4.2 shows the effect of this operation. In color images, same operation are applied into all three value of color space.

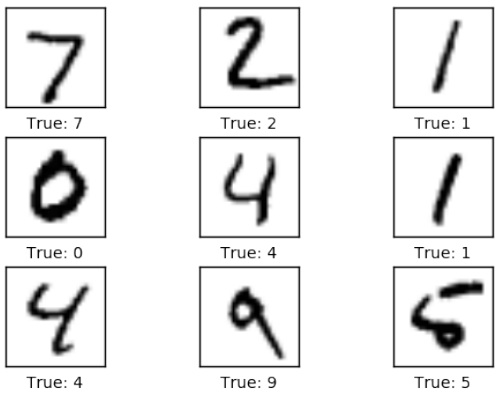


Figure 4.1: Original Image

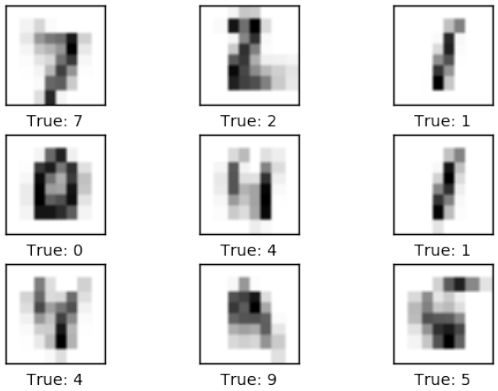


Figure 4.2: After Separating to 64 blocks

4.2.3 Canny Edge Detector

In order to extract Canny Edge Image, function ”edge()” on ”MATLAB Image Processing Toolbox” [20] was used. The following code shows the example of transforming color image to edge image.

```
I = imread('someColorImage.png');
GrayI = rgb2gray(I);
EdgeImage = edge(I,'Canny',threshold,sigma)
```

by changing ”threshold value” will control the strength of the edge that will surface in final edge image and sigma is the constant value for Gaussian smoothing filter.

4.2.4 Anisotropic Diffusion

Image Edge Enhancing Coherence Filter Toolbox [21] were used to create a edge enhanced image with anisotropic diffusion. First, execute **”compile_c_files.m”** to generate variables used later in the main process. Then, transform the image from RGB color space to HSV color space, then execute **”CoherenceFilter()”**(The author of the package suggested to use double type as image input, but in our experiments, using HSV color space as input performed better) Following is a sample way to call this function.

```
JO = CoherenceFilter(I,struct('T',15,'rho',10,'Scheme','O'));
```

The option ’T’ defines the diffusion time, ’rho’ defines the gaussian smoothing level

(a) Original Image (b) After Anisotropic Diffusion

Figure 4.3: The effect of Anisotropic Diffusion

that is applied through out the image, and 'scheme' defines the diffusion methods which we find 'O'(Optimised method) performed the best. Figure 4.3 shows the result image we get from this toolbox

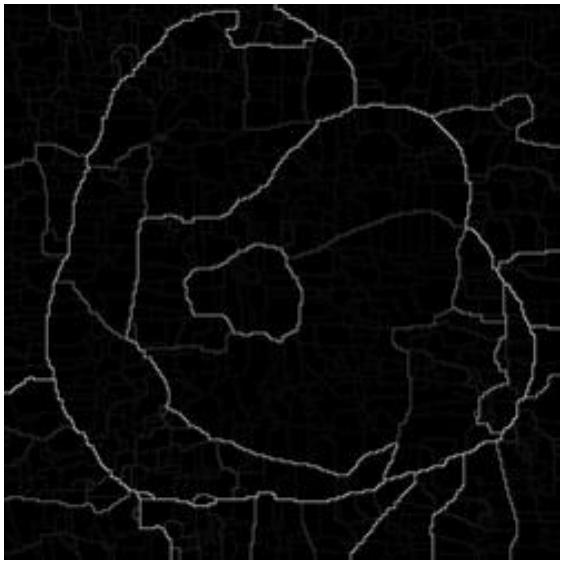
4.2.5 Contour Detection and Hierarchical Image Segmentation

The code for this project is being distributed on Original Project Page [22]

First run **"install.m"** inside directory "pre-trained". After this, excute function **"im2ucm()"** to extract Ultrametic Contour Map(UCM). there are two modes, 'fast' and 'accurate' in this function. The 'accurate' option took three times more than 'fast' and did not output much different contour, so we decided to use **"im2ucm(image,'fast')"**. Figure 4.4b shows the execution example of this.

However the direct contour image is hard to use, because all the other necessary information that neural network may catch are dissolved as well. Thus we decide to use the original image but with contour or important parts enhanced. In RGB space we can change RGB value individually, however changing each of the color of the

(a) Original Image



(b) UCM image

(c) x1.3 Brighter at Contour

(d) x1.3 More Saturated at Contour

Figure 4.4: Using Ultrametric Contour Map (UCM)

image will dynamically change the color correlations between pixels in the original image. We believe changing to HSV color space will solve this problem. In HSV color space, we can change saturation value or brightness which will not greatly reduce the color information of the original pixels. The result on Figure 4.4c shows a bright color around the contour of the flower. Figure 4.4d shows a highly saturated purple around the contour of the flower.

4.3 Neural Network Construction

4.3.1 TensorFlow

TensorFlow [11] is an open source software library for conducting machine learning tasks. It was released on late 2015 by Google Brain: a deep learning research project at Google. We used tensorflow as deep learning tool for the following reason.

- GPU support without explicitly writing parallel code
- Great support at user community
- Uses Python as API and C++ back-end.

4.3.2 Building Convolutional Neural Network

In this research, we implemented a 2 layer convolutional neural network. Figure 4.5 shows the specification of my network. Since this is a very simple network, it can not construct a very good model for the classification, but since the main of this research is the feature extractor and with smaller network we can test many kinds of feature extractors and different variabl for the extractors. The performance of the feature extractor mentioned in chapter 2 will be based on the accuracy output from this network.

The Source Code for this part can be found at Appendix A.1. It is based on the Code from Tensorflow tutorial [23] by Magnus Erik Hvass Pedersen. In this section we will go through the steps of implementing our convolutional neuron network.

First, we have Algorithm 2 that shows the process of defining a function for making new convolution layer **"new_conv_layer()"**. It will take 4 dimension tensor [num_images, img_height, img_width, num_channels] as Images, number of color channels, size of filter, number of filters and whether or not using pooling operation.

Second, we need to define a fully connected layer that comes right after the last convolution layer **"new_fc_layer()"**(Algorithm 3). The input for this function are Im-

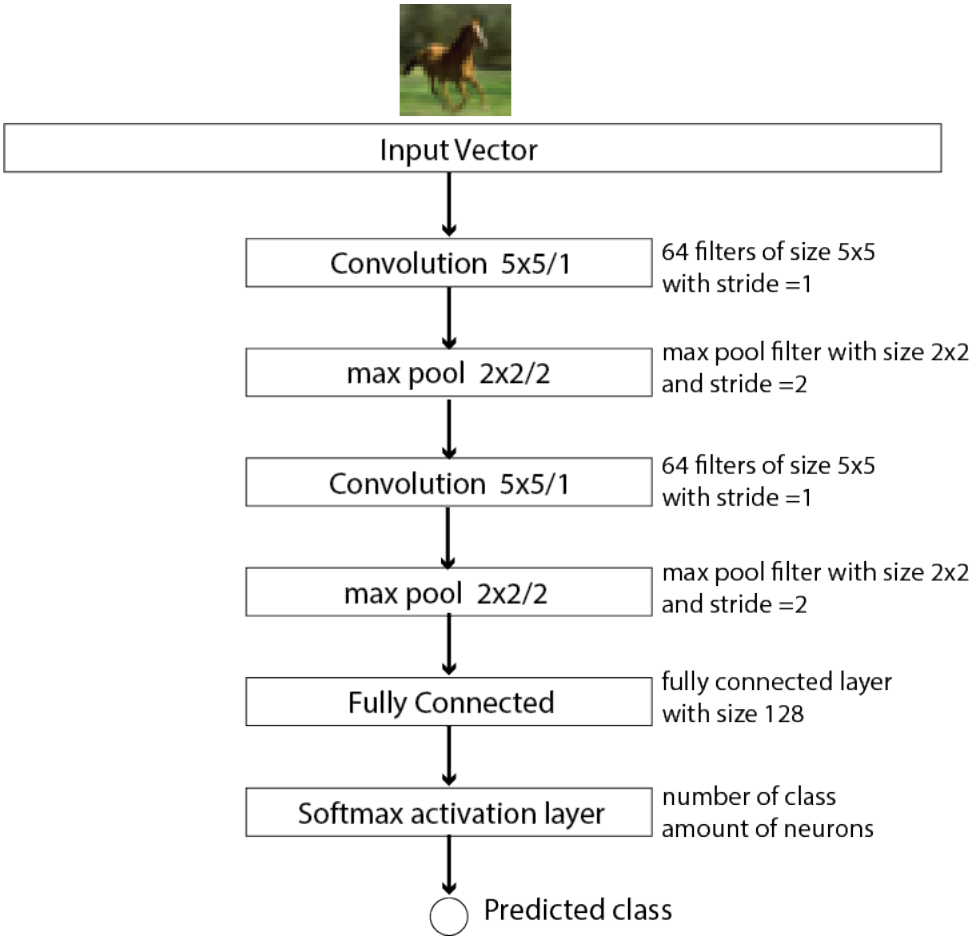


Figure 4.5: Convolutional Neural Netowrk used in this research

ages which are now 2 dimension matrix with shape=[num_images, num_features(=img_height * img_width * num_channels)], input channels (= num_features), number of output and whether or not using relu activation function.

Algorithm 3: Constructing Fully connected Layer **"new_fc_layer()"**

1. initialize **weights** as 0 with shape [number of inputs, number of outputs]
2. initialize **biases** as a vector with size of 'number of outputs'
3. define **layer** with function tf.matmul(input, weights) + biases
4. **If use_relu** is True:
   update **layer** with tf.nn.relu function

Data: input, numinput channels, number of outputs, use_relu
Result: output of the layer

The next step, Algorithm 4, is where construct a network using the two functions

Algorithm 2: Constructing Convolution Layer **"new_conv_layer()"**

1. define **tensor_shape** as [filter size, filter size, number of input channels, number of filters]
2. initialize **weights** as 0 with shape of tensor_shape
3. initialize **biases** as a vector with size of 'number of filters'
4. define **layer** with function tf.nn.conv2d
5. add **bias** to **layer**
6. **If use_pooling** is True:
   layer is updated with 2x2 tf.nn.max_pool function
7. update **layer** with tf.nn.relu function

Data: Images, num_input_channels, filter_size, num_filters, use_pooling
Result: output_layer, weights

defined above. it creates a network with 2 convolution layer and 1 fully connected layer, then use softmax function to evaluate the predicted class which are then combined with true label to generate cost, so we could use Gradient Descent algorithm (Adam Optimizer in this case) to reduce the cost.

the steps on Algorithm 5 involves the running of the network that we have constructed. **"tf.Session()"** are used to initialize a session that creates a network then use **session.run(tf.initialize_all_variables())** to initialize variables defined at the top of Algorithm 4.

Then we need to create a function that fetches new training samples called batch and pass to **"session.run()"** with **optimizer** defined in Algorithm 4. The batch is dictionary type variable which means it is the combination of the training images and it's true label.

Algorithm 4: Constructing Network of two convolution layer and two fully connected layer

1. initialize x, x_image, y_true, y_true_cls as tensor variable
2. call "new_conv_layer()" to construct convolution layer and return layer_conv1 and weights1 as output
3. call "new_conv_layer()" and pass layer_conv1 as input. It will construct convolution layer and return layer_conv2 and weights2 as output.
4. Flatten layer_conv2 from 4 dimension tensor to 2 dimension [num_images, num_features] and get layer_flat and num_features.
5. call "new_fc_layer()" with layer_flat as input and return layer_fc1
6. call "new_fc_layer()" with layer_fc1 as input and return layer_fc2
7. call softmax function to return the probability of which class does the prediction belongs.
8. use argmax() to return the class index that the network calculated as the most likely from the output of softmax function.
9. calculate the cost(cross_entropy) with function "softmax_cross_entropy_with_logits()" by giving layer_fc2 and y_true
10. Defining optimizer for reducing cost using "optimizer = AdamOptimizer(trainingRate).minimize(cost)"

Algorithm 5: Running the network

1. Start the session with network constructed above with **"session = tf.Session()"** and **"session.run(tf.initialize_all_variables())"**
2. construct a function that for every iteration it will fetch a random sample(feed_train) from training data and use it to run the optimizer. **"session.run(optimizer, feed_train)"**

By repeating the step 2 on algorithm 4, the network will be improved, so that it can adapt the training data. In this way, we can teach network to learn images or objects.

Algorithm 6: Constructing Fully connected Layer **"new_fc_layer()"**

1. initialize input image x as 4 dimension tensor.
2. <Encoder> For each layer initialize weight w, bias b as tensorflow variable. Then, apply $w \cdot x + b$ and activation function to return output z.(if it is convolution layer, apply convolution operation "tf.nn.conv2d"and then activation function)
3. <Decoder> For each layer initialize weight w' whose shape is transpose of w and bias b' as tensorflow variable. Then, apply $w' \cdot z + b'$ and activation function to return output y.(if it is convolution layer, apply convolution operation "tf.nn.conv2d_transpose" and then activation function)
4. define **cost** as the difference between y and x, and minimize the cost when starting this session.
5. return w and b for each layer

Data: input_shape, num_filter (as array), filter_sizes (as array)
Result: x, z, y, cost, w, b

4.3.3 Constructing Autoencoder

In autoencoer, as explained in chapter 2, we will first construct a similar network model, then define the **cost** to optimize it. we use input data as expected output data, and becomes a network that will reduce dimension in the inner layers and mimicking a images same as input. The brief algorithm can be written as Algorithm 6 and the source code at Appendix A.2.

4.3.4 Transfer learning with pretrained network

In order to make a practical use of neural network to classify objects, the network with only two convolution layers or even ten layers are not going to be enough. Even we have a good resource of hardware to construct a network, it still takes a long time to train a network with large number of classes. Instead, we could continue train the pretrained network that are already developed to suit our classification tasks.

This approach is called "Transfer Learning". With Transfer Learning we could train a complicated network with little computation resources and time to construct a highly accurate classifier.

In tensorflow, the easiest pretrained model to work on is "Inception-v3" model which was released simultaneously with tensorflow. Inception-v3 [24] is trained with ImageNet [25], a huge database of images with labels. It was trained as 1000-class classifications and has the network structure as on Figure 4.6.

There are two ways to make use of this pretrained network. One is to use the network as it is but change the last 2 classification layers. In the training process, variables inside the pretrained network will be kept as it is, but layer that were being added are being optimized to improve the accuracy of the prediction. The other involves optimizing or fine-tuning every weight and bias inside the network. This means that for every iteration we need to calculate the back-propagation though the whole network which is very difficult with our testing machine. Thus, natually, only the first method can be used in this research.

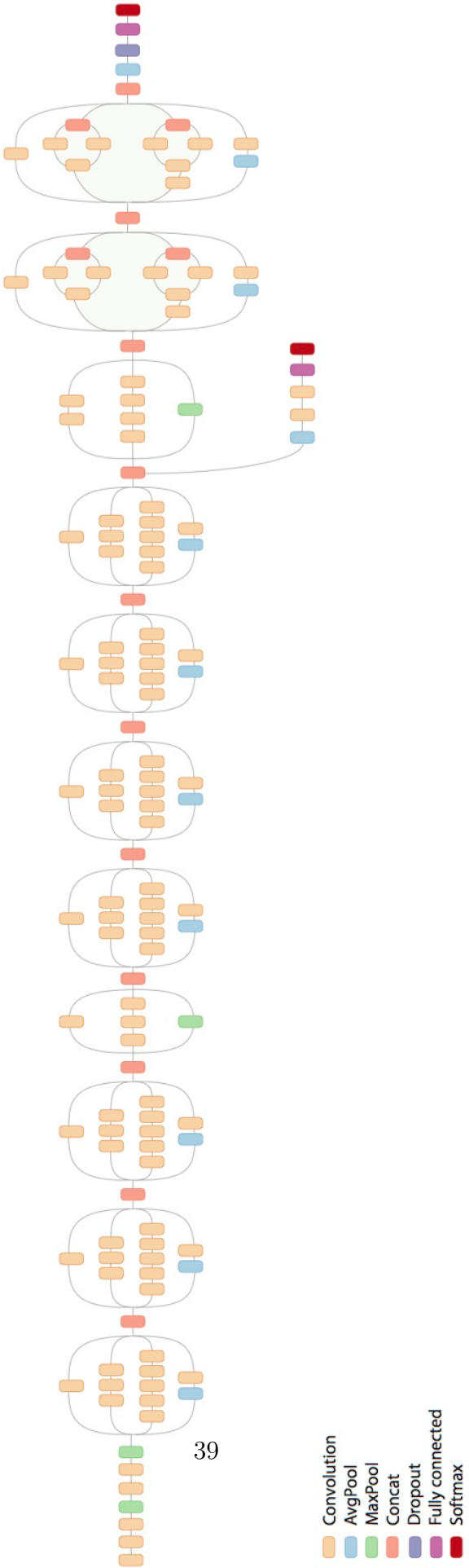


Figure 4.6: Google Inception Model

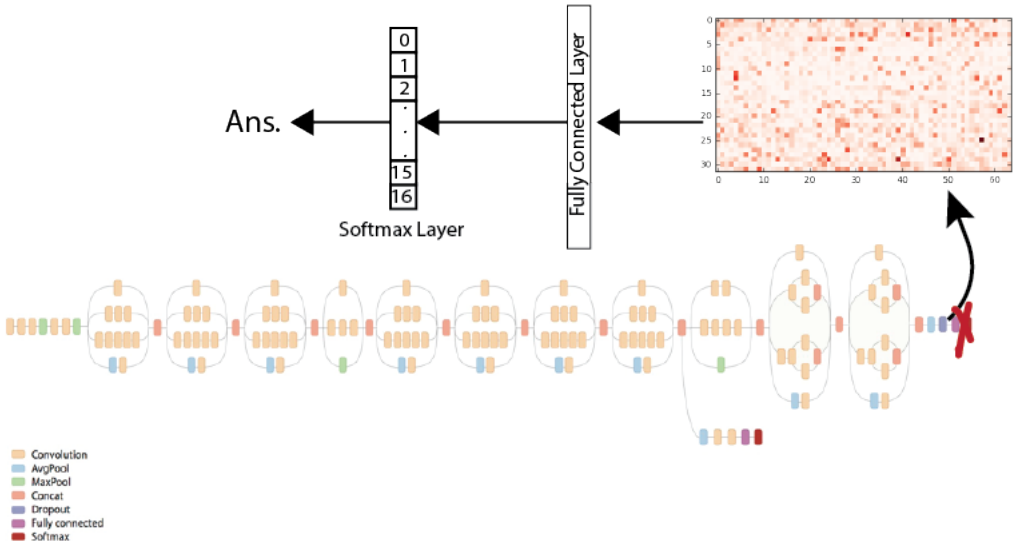


Figure 4.7: Transfer neural network that were used in this research

Figure 4.7 shows the process of transfer learning with flower dataset of 17 classes.

First we need to send all the images in the dataset to Inception-v3 model where we can have a raw transfer values. By analysing these values (Figure 4.8) with t-SNE , we find that the inception-v3 are capable of classifying input images into groups.

Then, we train a 2 fully connected network with transfer values as it's input and apply gradient descent to reduce the error with the expected output. This process takes really small time to compute because we only have 2 layers.

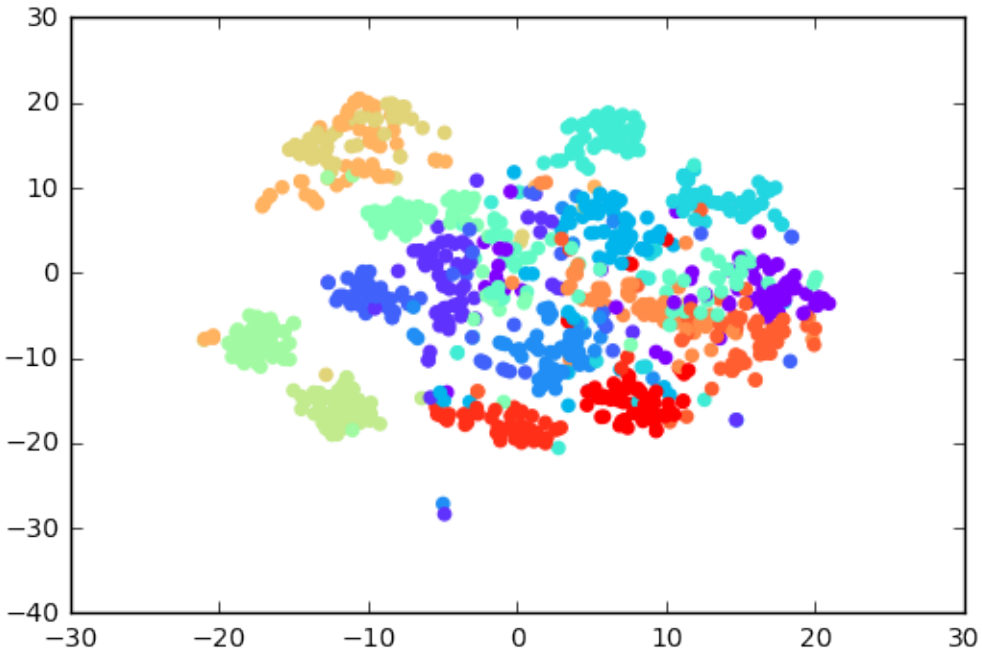


Figure 4.8: Analysis of the transfer value (50 plots each for each class)

4.4 Appling Feature Extracted Image in Neural Network

Now, we would combine the feature extracted image with neural network. There were two approach were taken in order to use the feature extracted images.

1. change the input images (for both training and testing) into feature extracted images.

2. use feature extracted image to train the autoencoder.

We have tested with both of the two methods. we could apply method 1 into both my network and inception model, but method 2 can only be tested with my simple network. This is because in order to use autoencoder, we need to create another network besides the main network, which will be really difficult for deep layers such as Inception model.

In the next chapter we will go through the result of these approach and find

whether feature extracting as the reprocess to the images are helping the network to reach more accurate prediction or not.

Chapter 5

Experiments and Discussion

5.1 Datasets

In this research, following four datasets were used.

1. MNIST
2. CIFAR10
3. 17 Flower Category Database
4. Custom Made Flower Datasets

5.2 Results on Each Dataset

5.2.1 MNIST

MNIST is a database of handwritten digits. it has 60,000 examples of training sets and 10,000 examples of test sets. Following shows some of example images of MNIST.

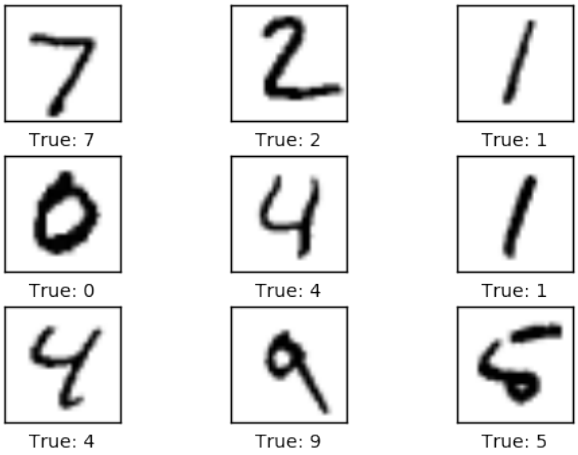


Figure 5.1: MNIST Images

Table 5.1: Excution Result on MNIST Datasets

MNIST	Accuracy	Iteration
Simple Linear Model	91.30%	1,000
CNN	98.70%	10,000
pretrain+ CNN	98.80%	10,000
Edge-Pretrain + CNN	98.90%	10,000
Edge-Pretrain + EdgeCNN	98.10%	10,000
CLD-CNN(Gray Blocks)	94.30%	10,000

Table 5.1 shows the result of test conducted on this datasets. First 2 rows shows a big improvements on accuracy when using convolutional neural network than feed-forward neural network. From row 3, pretraining using autoencoder is adopted before start the main session of learning. row 3 uses original image as pretrain which had 0.1% improvements over network without pretraining. In row 4, edge images (same as contour because the inside of letter is filled) are used in the process of pretraining and we achieve another 0.1% improvements compared with method of using original image at pretraining. We expected to get high accuracy on the condition that both the pretraining and actual training uses only edge images, however we get a slight drop of accuracy. On the other hand, Color Layout Descriptor's constructing 8x8 dominant color block, were tested on last row which showed a big drop of classification accuracy for over 4%. From the experiments on this datasets, we can conclude that shape or

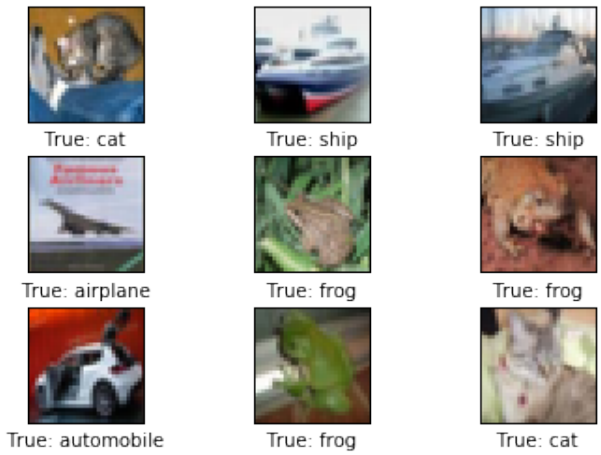


Figure 5.2: Sample CIFAR10 Images

Table 5.2: Excution Result on CIFAR10 Datasets

CIFAR10	Accuracy	Iteration
CNN	79.10%	100,000
pretrain+ CNN	29.30%	100,000
AD-pretrain + CNN	10.60%	100,000
Transfer Learning	90.70%	100,000

edge is an important characteristic inside a image rather than the abstract spacial color distribution.

5.2.2 CIFAR10 Data sets

CIFAR10 has 60,000 32x32 color images in 10 classes. 50,000 of them are training data and 10,000 are test data. The classes consists natural images of airplane, automobile, bird, cat,deer,dog,frog,horse, ship and truck. Since it is only 32x32, we expect the effect of the filter is small or even worsen because the information are too small to take away even more features. Following shows some of the example images of this data sets. 5.2 shows the result of test conducted on CIFAR10 Datasets. The first row uses convolutional neural network that we constructed in this project. the second row uses autoencoder as pretraining, however we get a huge accuracy loss of around 50 %. We believe that there are some flaws in our program that does not pretrain color image correctly. In row 3, anisotropic diffusion are used as the input image for pretraining

(a) Original Image (b) After Anisotropic Diffusion

Figure 5.3: Example of corrupted images after anisotropic diffusion

which has even bad result than the method using original image in pretraining. This is mainly due to the poor anisotropic diffusion result we get in our implementation. Figure 5.3 shows the example of corrupted images we get after anisotropic diffusion. The Anisotropic Diffusion not only failed to keep the edge of objects but also adding curve which destroyed the shape of objects. The last row uses Transfer Learning which uses Google's Inception model and it returned a very high 90% as accuracy.

5.2.3 Oxford 17 Flower Category Database

This data sets consists of 17 classes and 80 different images inside each class. It is really small compared with other data sets. Thus, in this research, we created a new data sets with more images but same classification as explained in next section.

Table 5.3: Excution Result on Oxford 17 Flower Category Database

OxfordFlowerDatasets	Accuracy	Iteration
CNN	58.80%	4,100
Transfer	95.20%	1,000
Saturation:x1.1 Transfer	96.00%	2,000
Saturation:x1.2 Transfer	95.60%	2,000
Saturation:x1.3 Transfer	95.60%	2,000
Saturation:x1.4 Transfer	95.20%	2,000
Saturation:x1.5 Transfer	95.60%	2,000

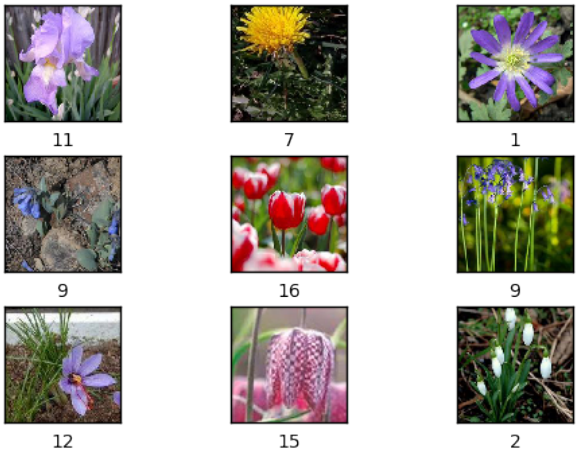


Figure 5.4: Sample Flowers Images

The precious test uses different feature suggested edge or boarder feature inside the image are highly likely to be able to improve the result of classification. Thus from this datasets we are focused on applying different methods to strengthen contour that found inside the image. As on Table 5.3, Saturation value changed at contour had a stable improvements to the accuracy as high as 0.8% improvement. This showed one of the sign that contour has some influence on the neural network and saturation is one of such method to help neural network use them as feature points.

5.2.4 Custom Made Flower Datasets

This custom handmade made data sets consists of same 17 classes but 401 different images inside each class. All the image is downloded from ImageNet [8]. We apply trimming and resizing to create 200x200 images from them.

In this datasets, the classification tasks becomes much harder than the previous one. This is because we have more variety of images to deal with which requires more iteration in order to adapt them. The accuracy with raw images reached around 84.1 %. With contour Saturation exaggerated images, the accuracy had improved by 0.4% but went back to the same accuracy when the saturation value at those points are weakened or strengthened for too much. This results and results on previous datasets

Table 5.4: Excution Result on Custom Made Flower Datasets

NewFlowerDatasets	Accuracy	Iteration
Transfer	84.10%	4,300
Sat:x0.9 Transfer	84.10%	4,300
Sat:x1.1 Transfer	84.50%	4,300
Sat:x1.3 Transfer	84.10%	4,300

has one similarity of achieving hiest accuracy improvements when contour saturation value were multiplied by 1.1.

5.3 Effect on hidden layers of neural network

We can see the effect of the feature enhanced images by print out the layer output of the hidden layers. We used a simple convolutional neural network described in chapter 4 and trained it with Oxford 17 Flower Category Datasets. With Figure 5.5 as input, figure 5.6 show the output of the first convolutional layer with non-enhanced image. The other (Figure 5.7) is the output with enhanced image. Both are trained in their own datasets. From these figure, we can find more contour like image or object than the output using non-enhanced features as input.

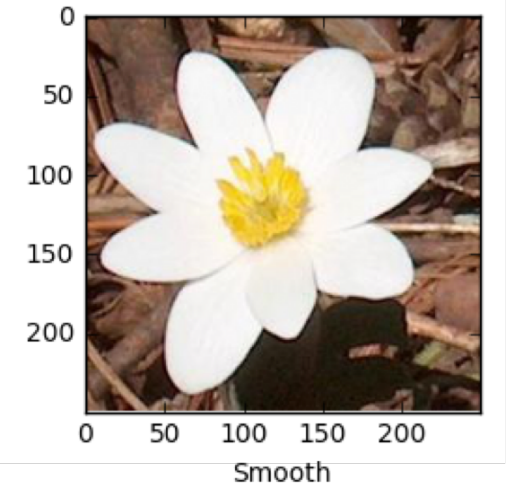


Figure 5.5: the output of the first convolution layer with non-enhanced image

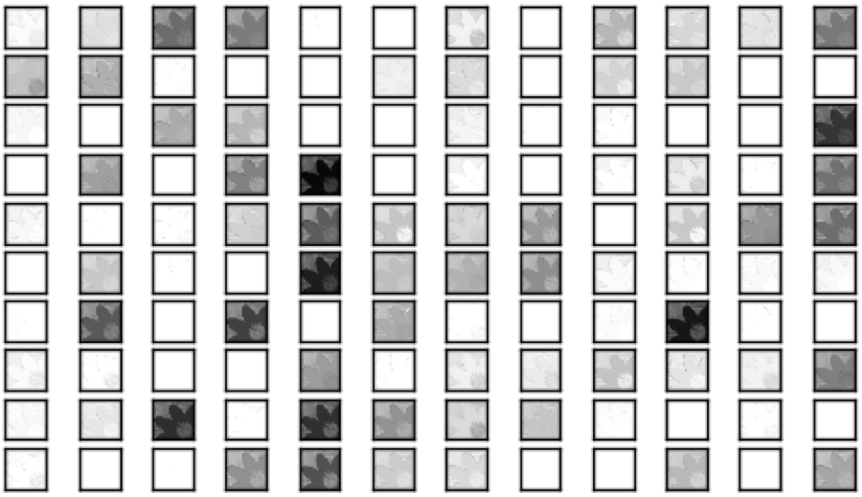


Figure 5.6: the output of the first convolution layer with non-enhanced image

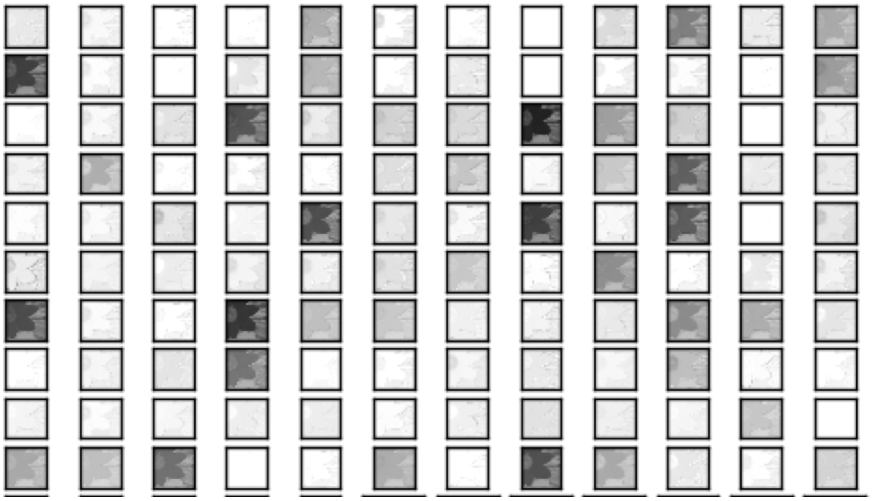


Figure 5.7: output of the first convolution layer with enhanced images

Same effect can be seen on second convolution layer as 5.8 and 5.9

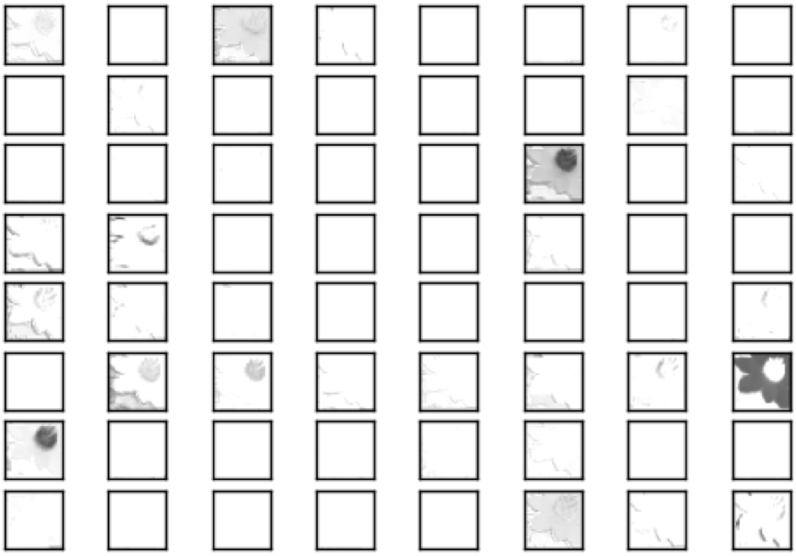


Figure 5.8: the output of the second convolution layer with non-enhanced image

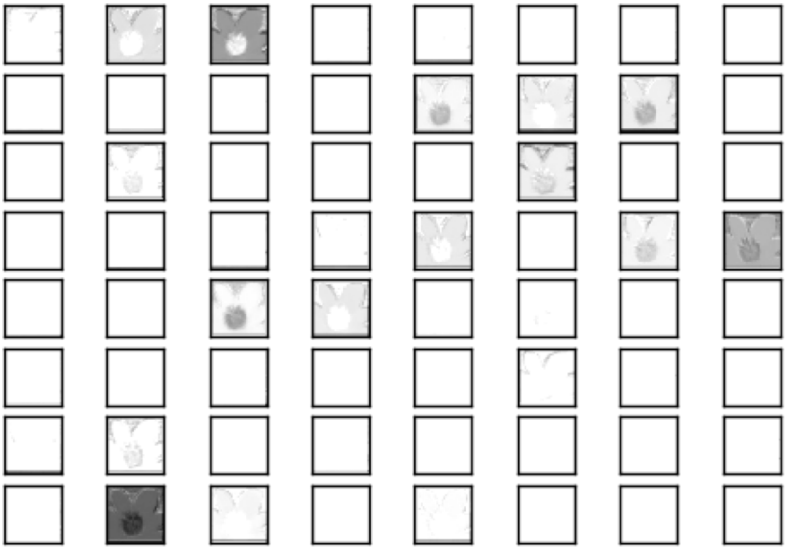


Figure 5.9: output of the second convolution layer with enhanced images

From the results of classification accuracy and the layer output of hidden layer, we can conclude that we have successfully altered the behavior of the neural network to focus on feature, contour in this case, that we wished them to focus.

Chapter 6

Conclusion

This research was started in mind that neural network alone is not good enough to extract useful and important feature of the images. We know from the analysis in MPEG-7 that color, texture and shape are the main features that we can be detected by computer. We thought it would be interesting to find out what happen, if we use highly sophisticated feature extractor and apply those images into the neural network. As chapter 5 shows, The contour or shape of the object are actually very important feature in neural network as well and strengthen them had brought even more accurate classification on objects. Through the experiments, we also found that changing of saturation values are one of the practical way to make neural network focus on feature that we want it to prioritizes.

However, this research has far more things to do in the future. First, the amount of experiments are too few to conclude that the contour saturation method works in variety kinds of natural images. Second, feature such as color and texture are not studied in this research. Good Anisotropic Diffusion implementation that makes texture blurred and contour saturated was one of the feature extractor that combines color and shape. Third is the neural network model should be updated to reflect the fast evolving recent research on neural network.

As of right now, we plan to prioritize on implementing the implementation of our own anisotropic diffusion since rather than only saturate the boarder to make

contour more attractive, we also want to make other part less attractive. We believe anisotropic diffusion or the combination with contour extracted from Ultra-metric Contour Map can create a image that uses color feature and shape feature. By having more core features exaggerated inside a picture, we believe the classification accuracy using neural network will increase even more.

References

[1] Michael Bernstein Yuke Zhu Oliver Groth and Li Fei-Fei. "Grounded Question Answering in Images". In: (2016). arXiv: 1511.03416 [quant-ph].

[2] Michael Stark Justin Johnson Ranjay Krishna et al. "Scene Graph Generation by Iterative Message Passing". In: (2015). arXiv: 1701.02426 [quant-ph].

[3] Pedro F. Felzenszwalb et al. "Object Detection with Discriminatively Trained Part-Based Models". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 32.9 (Sept. 2010), pp. 1627–1645. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2009.167. URL: http://dx.doi.org/10.1109/TPAMI.2009.167.

[4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[5] Alex Krizhevsky. "Learning Multiple Layers of Features from Tiny Images". In: (2009). URL: https://www.cs.toronto.edu/~kriz/cifar.html.

[6] L. Bo et al. "Object recognition with hierarchical kernel descriptors". In: *CVPR 2011*. June 2011, pp. 1729–1736. DOI: 10.1109/CVPR.2011.5995719.

[7] Benjamin Graham. "Fractional Max-Pooling". In: *CoRR* abs/1412.6071 (2014). URL: http://arxiv.org/abs/1412.6071.

[8] Jia Deng Wei Liu Olga Russakovsky, Fei-Fei Li, and Alex Berg. "Large Scale Visual Recognition Challenge (ILSVRC) 2016". In: (2016). URL: http://image-net.org/challenges/talks/2016/ECCV2016_ilsvrc_coco_detection_segmentation.pdf.

[9] Thomas Sikora B. S. Manjunath Philippe Salembier. *Introduction to MPEG-7: Multimedia Content Description Interface*. Wiley, 2002.

[10] Y. Le Cun et al. "Advances in Neural Information Processing Systems 2". In: ed. by David S. Touretzky. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990. Chap. Handwritten Digit Recognition with a Back-propagation Network, pp. 396–404. ISBN: 1-55860-100-7. URL: http://dl.acm.org/citation.cfm?id=109230.109279.

[11] Google. *Tensorflow*. 2015. URL: https://www.tensorflow.org.

[12] Yangqing Jia Christian Szegedy Wei Liu et al. “Going Deeper with Convolutions”. In: (2014). arXiv: 1409.4842 [quant-ph].

[13] B. S. Manjunath et al. “Color and texture descriptors”. In: *IEEE Transactions on Circuits and Systems for Video Technology* 11.6 (June 2001), pp. 703–715. ISSN: 1051-8215. DOI: 10.1109/76.927424.

[14] P. Perona and J. Malik. “Scale-Space and Edge Detection Using Anisotropic Diffusion”. In: *IEEE Trans. Pattern Anal. Mach. Intell.* 12.7 (July 1990), pp. 629–639. ISSN: 0162-8828. DOI: 10.1109/34.56205. URL: http://dx.doi.org/10.1109/34.56205.

[15] P. Arbelaez. “Boundary Extraction in Natural Images Using Ultrametric Contour Maps”. In: *2006 Conference on Computer Vision and Pattern Recognition Workshop (CVPRW'06)*. June 2006, pp. 182–182. DOI: 10.1109/CVPRW.2006.48.

[16] Manasi Datar. *Project 2: Anisotropic Diffusion*. URL: http://www.cs.utah.edu/~manasi/coursework/cs7960/p2/project2.html.

[17] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[18] Takayuki Okatani. *Kikaigakusyu Professional Series, Shinsougakusyu*. Koudansya, 2015.

[19] Quoc V. Le. *A Tutorial on Deep Learning Part 2: Autoencoders, Convolutional Neural Networks and Recurrent Neural Networks*. Oct. 2015. URL: https://cs.stanford.edu/~quocle/tutorial2.pdf.

[20] MathWorks. *MATLAB Image Processing Toolbox*. URL: https://www.mathworks.com/products/image.html.

[21] Dirk-Jan Kroon. *Image Edge Enhancing Coherence Filter Toolbox*. 2010. URL: https://https://www.mathworks.com/matlabcentral/fileexchange/25449-image-edge-enhancing-coherence-filter-toolboxl.

[22] Pablo Arbelaez et al. *Contour Detection and Image Segmentation Resources*. 2011. URL: https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/resources.html.

[23] Magnus Erik Hvass Pedersen. *TensorFlow Tutorials*. 2016. URL: https://github.com/Hvass-Labs/TensorFlow-Tutorials.git.

[24] Christian Szegedy et al. “Rethinking the Inception Architecture for Computer Vision”. In: *CoRR* abs/1512.00567 (2015). URL: http://arxiv.org/abs/1512.00567.

[25] Olga Russakovsky et al. “ImageNet Large Scale Visual Recognition Challenge”. In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y.

Appendices

Appendix A

Source Codes

Since there are so many code that I used, it is difficult to fit all in this appendix. Instead I had posted all the code in this thesis on the online repository at https://github.com/WaratenZ/Thesis-programs.git. I will list three programs that were the core to this project.

A.1 Convolutional Neural Network with CIFAR-10 images

```
#########################################################################
# # Author: Xiaotian Zhao
# The following program is modified based on code '06_CIFAR-10.ipynb'
# from https://github.com/Hvass-Labs/TensorFlow-Tutorials.git
#########################################################################
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
from sklearn.metrics import confusion_matrix
import time
from datetime import timedelta
import math
import os
import prettytensor as pt
# Import training data
import cifar10 # cifar10.py need to be on the same directory

cifar10.data_path = "data/CIFAR-10/"
file_path = cifar10._get_file_path()
class_names = cifar10.load_class_names()
images_train, cls_train, labels_train = cifar10.load_training_data()
images_test, cls_test, labels_test = cifar10.load_test_data()

# See What is inside
```

```
print("Size of:")
print ("- Training-set: \t\t{}".format(len(images_train)))
print ("- Training-set: \t\t{}".format(len(images_test)))

from cifar10 import img_size, num_channels, num_classes
img_size_cropped = 24

# Define function to some sample plot images
def plot_images(images, cls_true, cls_pred=None, smooth=True):
    assert len(images) == len(cls_true) ==9

    fig, axes = plt.subplots(3,3)

    if cls_pred is None:
        hspace = 0.3
    else:
        hspace = 0.6

    fig.subplots_adjust(hspace=hspace, wspace=0.3)

    for i, ax in enumerate(axes.flat):
        if smooth:
            interpolation = 'spline16'
        else:
            interpolation = 'nearest'

        ax.imshow(images[i,:,:,:],
                  interpolation = interpolation)

        cls_true_name = class_names[cls_true[i]]

        if cls_pred is None:
            xlabel = "True: {0}".format(cls_true_name)
        else:
            cls_pred_name = class_names[cls_pred[i]]
            xlabel = "True: {0}\nPred: {1}".format(cls_true_name,
    cls_pred_name)
        ax.set_xlabel(xlabel)

        ax.set_xticks([])
        ax.set_yticks([])

    plt.show()

# plot the sample images
images = images_test[0:9]
cls_true = cls_test[0:9]
plot_images(images=images, cls_true=cls_true, smooth=False)
# WITH SMOOTHING
plot_images(images=images, cls_true=cls_true, smooth=True)

# ## Constructing TensorFlow Graph
x = tf.placeholder(tf.float32, shape=[None, img_size, img_size,
    num_channels], name='x')
```

```
77 y_true = tf.placeholder(tf.float32, shape=[None, num_classes],name='
   y_true')
78 y_true_cls = tf.argmax(y_true, dimension=1)
79
80
81 # ### preprocessing image
82 # Adding variety on the imput images
83 def pre_process_image(image, training):
84     if training:
85         # Ransomly crop the input image
86         image = tf.random_crop(image, size=[img_size_cropped,
   img_size_cropped, num_channels])
87
88         image = tf.image.random_flip_left_right(image)
89         image = tf.image.random_hue(image, max_delta=0.05)
90         image = tf.image.random_contrast(image,lower=0.3, upper = 1.0)
91         image = tf.image.random_brightness(image, max_delta=0.2)
92         image = tf.image.random_saturation(image, lower=0.0, upper=2.0)
93
94         ## we limit the range of the value of image because some of the
   image\
95         ## is overflowed or underflowed
96
97         image = tf.minimum(image, 1.0)
98         image = tf.maximum(image, 0.0)
99
100     else:
101         image = tf.image.resize_image_with_crop_or_pad(image,
102                                                        target_height=
   img_size_cropped,
103                                                        target_width =
   img_size_cropped)
104
105     return image
106
107
108 def pre_process(images, training):
109     images = tf.map_fn(lambda image: pre_process_image(image, training),
   images)
110
111     return images
112
113 distorted_images = pre_process(images=x, training=True)
114
115 def conv2d():
116     return conv2d(kernel=5, depth=64, name='layer_conv1',
   batch_normalize=True)
117
118
119 # In[18]:
120
121 def main_network(images, training):
122     # Wrap the input images as a Pretty Tensor object.
123     x_pretty = pt.wrap(images)
```

```
124
125     if training:
126         phase = pt.Phase.train
127     else:
128         phase = pt.Phase.infer
129
130     # Create the convolutional neural network using Pretty Tensor.
131     # It is very similar to the previous tutorials, except
132     # the use of so-called batch-normalization in the first layer.
133     with pt.defaults_scope(activation_fn=tf.nn.relu, phase=phase):
134         y_pred, loss = x_pretty.                conv2d(kernel=5, depth=64,
    name='layer_conv1', batch_normalize=True).              max_pool(kernel
    =2, stride=2).              conv2d(kernel=5, depth=64, name='
    layer_conv2').              max_pool(kernel=2, stride=2).
    flatten().              fully_connected(size=256, name='layer_fc1').
              fully_connected(size=128, name='layer_fc2').
    softmax_classifier(num_classes, labels=y_true)
135
136     return y_pred, loss
137
138
139 ### Create network for training
140
141 def create_network(training):
142     # Wrap the neural network in the scope named 'network'.
143     # Create new variables during training, and re-use during testing.
144     with tf.variable_scope('network', reuse=not training):
145         # Just rename the input placeholder variable for convenience.
146         images = x
147
148         # Create TensorFlow graph for pre-processing.
149         images = pre_process(images=images, training=training)
150
151         # Create TensorFlow graph for the main processing.
152         y_pred, loss = main_network(images=images, training=training)
153
154     return y_pred, loss
155
156 global_step = tf.Variable(initial_value=0,
157                           name='global_step', trainable=False)
158 _, loss = create_network(training=True)
159
160 optimizer = tf.train.AdamOptimizer(learning_rate=1e-4).minimize(loss,
161
    global_step=global_step)
162
163
164 ### Create Network for test phrase
165
166 y_pred, _ = create_network(training=False)
167 y_pred_cls = tf.argmax(y_pred, dimension=1)
168 correct_prediction = tf.equal(y_pred_cls, y_true_cls)
169 accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
170 ## Saver
```

```
saver = tf.train.Saver()
def get_weights_variable(layer_name):
    with tf.variable_scope("network/" + layer_name, reuse=True):
        variable = tf.get_variable('weights')

    return variable

weights_conv1 = get_weights_variable(layer_name='layer_conv1')
weights_conv2 = get_weights_variable(layer_name='layer_conv2')


### get the layer output

def get_layer_output(layer_name):
    # The name of the last operation of the convolutional layer.
    # This assumes you are using Relu as the activation-function.
    tensor_name = "network/" + layer_name + "/Relu:0"

    # Get the tensor with this name.
    tensor = tf.get_default_graph().get_tensor_by_name(tensor_name)

    return tensor


output_conv1 = get_layer_output(layer_name='layer_conv1')
output_conv2 = get_layer_output(layer_name='layer_conv2')

session = tf.Session()

save_dir = 'checkpoints/'

if not os.path.exists(save_dir):
    os.makedirs(save_dir)

save_path = save_dir + 'cifar10_cnn'

try:
    print("Trying to restore last checkpoint")
    last_chk_path = tf.train.latest_checkpoint(checkpoint_dir=save_dir)
    # use tensorflow function to find last checkpoint

    saver.restore(session, save_path= last_chk_path)
    print("Restored checkpoint from:", last_chk_path)

except:
    # If the above failed for some reason, simply
    # initialize all the variables for the TensorFlow graph.
    print("Failed to restore checkpoint. Initializing variables instead.
    ")
    session.run(tf.initialize_all_variables())

train_batch_size = 64

def random_batch():
```

```
    num_images = len(images_train)
    idx = np.random.choice(num_images,
                           size=train_batch_size,
                           replace=False)
    # Use the random index to select random images and labels.
    x_batch = images_train[idx, :, :, :]
    y_batch = labels_train[idx, :]

    return x_batch, y_batch

def optimize(num_iterations):
    start_time = time.time()

    for i in range(num_iterations):
        # Get a batch of training examples.
        # x_batch now holds a batch of images and
        # y_true_batch are the true labels for those images.
        x_batch, y_true_batch = random_batch()

        # Put the batch into a dict with the proper names
        # for placeholder variables in ttihe TensorFlow graph.
        feed_dict_train = {x: x_batch,
                           y_true: y_true_batch}

        # Run the optimizer using this batch of training data.
        # TensorFlow assigns the variables in feed_dict_train
        # to the placeholder variables and then runs the optimizer.
        # We also want to retrievie the global_step counter.
        i_global, _ = session.run([global_step, optimizer],
                                  feed_dict=feed_dict_train)

        # Print status to screen every 100 iterations (and last).
        if (i_global % 100 == 0) or (i == num_iterations - 1):
            # Calculate the accuracy on the training-batch.
            batch_acc = session.run(accuracy,
                                    feed_dict=feed_dict_train)

            # Print status.
            msg = "Global Step: {0:>6}, Training Batch Accuracy:
    {1:>6.1%}"
            print(msg.format(i_global, batch_acc))

        # Save a checkpoint to disk every 1000 iterations (and last).
        if (i_global % 1000 == 0) or (i == num_iterations - 1):
            # Save all variables of the TensorFlow graph to a
            # checkpoint. Append the global_step counter
            # to the filename so we save the last several checkpoints.
            saver.save(session,
                       save_path=save_path,
                       global_step=global_step)

            print("Saved checkpoint.")

    # Ending time.
```

```
277     end_time = time.time()
278 
279     # Difference between start and end-times.
280     time_dif = end_time - start_time
281 
282     # Print the time-usage.
283     print("Time usage: " + str(timedelta(seconds=int(round(time_dif)))))
284 
285 
286 # In[38]:
287 
288 def plot_example_errors(cls_pred, correct):
289 
290     incorrect = (correct == False)
291 
292     images = images_test[incorrect]
293 
294     cls_pred = cls_pred[incorrect]
295     cls_true = cls_test[incorrect]
296 
297     # Plot the first 9 images.
298     plot_images(images=images[0:9],
299                 cls_true=cls_true[0:9],
300                 cls_pred=cls_pred[0:9])
301 
302 
303 # In[39]:
304 
305 def plot_confusion_matrix(cls_pred):
306     # This is called from print_test_accuracy() below.
307 
308     # cls_pred is an array of the predicted class-number for
309     # all images in the test-set.
310 
311     # Get the confusion matrix using sklearn.
312     cm = confusion_matrix(y_true=cls_test,  # True class for test-set.
313                           y_pred=cls_pred)  # Predicted class.
314 
315     # Print the confusion matrix as text.
316     for i in range(num_classes):
317         # Append the class-name to each line.
318         class_name = "({}) {}".format(i, class_names[i])
319         print(cm[i, :], class_name)
320 
321     # Print the class-numbers for easy reference.
322     class_numbers = [" ({0})".format(i) for i in range(num_classes)]
323     print("".join(class_numbers))
324 ### Calculating classification
325 
326 batch_size= 256
327 def predict_cls(images, labels, cls_true):
328     num_images = len(images)
329     cls_pred = np.zeros(shape=num_images, dtype=np.int)
330 
```

```
    i=0
    while i < num_images:
        # The ending index for the next batch is denoted j.
        j = min(i + batch_size, num_images)

        # Create a feed-dict with the images and labels
        # between index i and j.
        feed_dict = {x: images[i:j, :],
                     y_true: labels[i:j, :]}

        # Calculate the predicted class using TensorFlow.
        cls_pred[i:j] = session.run(y_pred_cls, feed_dict=feed_dict)

        # Set the start-index for the next batch to the
        # end-index of the current batch.
        i = j

    # Create a boolean array whether each image is correctly classified.
    correct = (cls_true == cls_pred)

    return correct, cls_pred

def predict_cls_test():
    return predict_cls(images = images_test,
                       labels = labels_test,
                       cls_true = cls_test)

def classification_accuracy(correct):
    # When averaging a boolean array, False means 0 and True means 1.
    # So we are calculating: number of True / len(correct) which is
    # the same as the classification accuracy.

    # Return the classification accuracy
    # and the number of correct classifications.
    return correct.mean(), correct.sum()


### function to show the performance
def print_test_accuracy(show_example_errors=False,
                        show_confusion_matrix=False):

    # For all the images in the test-set,
    # calculate the predicted classes and whether they are correct.
    correct, cls_pred = predict_cls_test()

    # Classification accuracy and the number of correct classifications.
    acc, num_correct = classification_accuracy(correct)

    # Number of images being classified.
    num_images = len(correct)

    # Print the accuracy.
    msg = "Accuracy on Test-Set: {0:.1%} ({1} / {2})"
    print(msg.format(acc, num_correct, num_images))
```

```
385
386     # Plot some examples of mis-classifications, if desired.
387     if show_example_errors:
388         print("Example errors:")
389         plot_example_errors(cls_pred=cls_pred, correct=correct)
390
391     # Plot the confusion matrix, if desired.
392     if show_confusion_matrix:
393         print("Confusion Matrix:")
394         plot_confusion_matrix(cls_pred=cls_pred)
395
396
397 # ### plot example distoreted images
398 def plot_distorted_image(image, cls_true):
399     # Repeat the input image 9 times.
400     image_duplicates = np.repeat(image[np.newaxis, :, :, :], 9, axis=0)
401
402     # Create a feed-dict for TensorFlow.
403     feed_dict = {x: image_duplicates}
404
405     # Calculate only the pre-processing of the TensorFlow graph
406     # which distorts the images in the feed-dict.
407     result = session.run(distorted_images, feed_dict=feed_dict)
408
409     # Plot the images.
410     plot_images(images=result, cls_true=np.repeat(cls_true, 9))
411
412
413 def get_test_image(i):
414     return images_test[i, :, :, :], cls_test[i]
415
416 img, cls = get_test_image(16)
417
418 plot_distorted_image(img, cls)
419
420
421 optimize(num_iterations=100000)
422 print_test_accuracy(show_example_errors=True,
423                     show_confusion_matrix=True)
```

Convolutional Neural Network with CIFAR-10 images

A.2 Pretraining with autoencoder using edge images and convolutional neural netowork on MNIST Datasets

```
1
2 ########################################################################
3 # Author: Xiaotian Zhao
4 # This program is modified based on code
5 # '02_Convolutional_Neural_Network.ipynb'
```

```
# from https://github.com/Hvass-Labs/TensorFlow-Tutorials.git
#
########################################################################

import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
from sklearn.metrics import confusion_matrix
from skimage import feature
import time
from datetime import timedelta
import math

filter_size1 = 5          # Convolution filters are 5 x 5 pixels.
num_filters1 = 16  ##can change later       # There are 16 of these
    filters.

# Convolutional Layer 2.
filter_size2 = 5          # Convolution filters are 5 x 5 pixels.
num_filters2 = 36  ##can change later       # There are 36 of these
    filters.

# Fully-connected layer.
fc_size = 128             # Number of neurons in fully-connected layer.

from tensorflow.examples.tutorials.mnist import input_data
data = input_data.read_data_sets('data/MNIST/', one_hot=True)

print("Size of:")
print("- Training-set:\t\t{}".format(len(data.train.labels)))
print("- Test-set:\t\t{}".format(len(data.test.labels)))
print("- Validation-set:\t{}".format(len(data.validation.labels)))

data.test.cls = np.argmax(data.test.labels, axis=1)

img_size = 28

# Images are stored in one-dimensional arrays of this length.
img_size_flat = img_size * img_size

# Tuple with height and width of images used to reshape arrays.
img_shape = (img_size, img_size)

# Number of colour channels for the images: 1 channel for gray-scale.
num_channels = 1

# Number of classes, one class for each of 10 digits.

num_classes = 10

def plot_images(images, cls_true, cls_pred=None):
    assert len(images) == len(cls_true) == 9

    # Create figure with 3x3 sub-plots.
```

```
    fig, axes = plt.subplots(3, 3)
    fig.subplots_adjust(hspace=0.3, wspace=0.3)

    for i, ax in enumerate(axes.flat):
        # Plot image.
        ax.imshow(images[i].reshape(img_shape), cmap='binary')

        # Show true and predicted classes.
        if cls_pred is None:
            xlabel = "True: {0}".format(cls_true[i])
        else:
            xlabel = "True: {0}, Pred: {1}".format(cls_true[i], cls_pred
    [i])

        # Show the classes as the label on the x-axis.
        ax.set_xlabel(xlabel)

        # Remove ticks from the plot.
        ax.set_xticks([])
        ax.set_yticks([])

    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()


### Plot a few images# Get the first images from the test-set.
images = data.test.images[0:9]
print(images.shape)
# Get the true classes for those images.
cls_true = data.test.cls[0:9]

# Plot the images and labels using our helper-function above.
plot_images(images=images, cls_true=cls_true)


## Edge Extracted Image
edge_img = []
st_time = time.time()


for img_index in range(len(data.train.images)):
    now_image = data.train.images[img_index]
    now_edge_img = feature.canny(now_image.reshape(img_shape))
    edge_img.append(now_edge_img.reshape(img_size_flat))
    if(np.mod(img_index,1000) == 0):
        print ('image:',img_index)
fi_time = time.time()
print('time usage',str(timedelta(seconds=int(round(fi_time - st_time))))
    )

# Get the first images from the test-set.
images = edge_img[0:9]

```

```
# Get the true classes for those images.
cls_true = data.test.cls[0:9]

# Plot the images and labels using our helper-function above.
plot_images(images=images, cls_true=cls_true)


# ## TensorFlow Graph

def new_weights(shape):
    return tf.Variable(tf.truncated_normal(shape, stddev=0.05))


# In[16]:

def new_biases(length):
    return tf.Variable(tf.constant(0.05, shape=[length]))


# In[17]:

def autoencoder(input_shape=[None, img_size, img_size, num_channels],
                n_filters=[3, 64, 64],
                filter_sizes=[5, 5],
                corruption=False):
    # %%
    # input to the network
    x = tf.placeholder(
        tf.float32, input_shape, name='x')

    # %%
    # ensure 2-d is converted to square tensor.
    if len(x.get_shape()) == 2:
        x_dim = np.sqrt(x.get_shape().as_list()[1])
        if x_dim != int(x_dim):
            raise ValueError('Unsupported input dimensions')
        x_dim = int(x_dim)
        x_tensor = tf.reshape(
            x, [-1, x_dim, x_dim, n_filters[0]])
    elif len(x.get_shape()) == 4:
        x_tensor = x
    else:
        raise ValueError('Unsupported input dimensions')
    current_input = x_tensor

    # %%
    # Optionally apply denoising autoencoder
    #     if corruption:
    #         current_input = corrupt(current_input)

    # %%
    # Build the encoder
    encoder = []
    shapes = []
```

```
    biases = []
    for layer_i, n_output in enumerate(n_filters[1:]):
        n_input = current_input.get_shape().as_list()[3]
        shapes.append(current_input.get_shape().as_list())
        W = tf.Variable(
            tf.random_uniform([
                filter_sizes[layer_i],
                filter_sizes[layer_i],
                n_input, n_output],
                -1.0 / math.sqrt(n_input),
                1.0 / math.sqrt(n_input)))
        b = tf.Variable(tf.zeros([n_output]))
        biases.append(b)
        encoder.append(W)
        output = tf.add(tf.nn.conv2d(
                current_input, W, strides=[1, 2, 2, 1], padding='SAME'),
    b)
        current_input = output

    # %%
    # store the latent representation
    z = current_input
    encoder.reverse()  # because we want to deal with inner deep layer
    than outer layer and so on.
    biases.reverse()
    shapes.reverse()

    # %%
    # Build the decoder using the same weights
    for layer_i, shape in enumerate(shapes):
        W = encoder[layer_i]
        b = tf.Variable(tf.zeros([W.get_shape().as_list()[2]]))
        output = tf.add(
            tf.nn.conv2d_transpose(
                current_input, W,
                tf.pack([tf.shape(x)[0], shape[1], shape[2], shape[3]]),
                strides=[1, 2, 2, 1], padding='SAME'), b)

        current_input = output

    # %%
    # now have the reconstruction through the network
    y = current_input
    # cost function measures pixel-wise difference
    cost = tf.reduce_sum(tf.square(y - x_tensor))
    encoder.reverse()  # return to the original order for the output
    biases.reverse()
    # %%
    return {'x': x, 'z': z, 'y': y, 'cost': cost, 'weights2d': encoder,
            'biases2d': biases}


# ### Helper-function for creating a new Convolutional Layer
### RANDOMLY TO GENERATE BATCH
```

```
def random_batch(batch_size,img_data):
    x_batch = []
    num_images = len(img_data)
    idx = np.random.randint(num_images,
                            size = batch_size)
    for i in idx:
        x_batch.append(img_data[i][:])
    return x_batch


# In[19]:

ae = autoencoder(input_shape=[None, 784],
                 n_filters=[1, 16, 36],
                 filter_sizes=[5, 5],
                 corruption=False)
mean_img = np.mean(data.train.images, axis=0)
learning_rate_pre = 0.01
optimizer_pre = tf.train.AdamOptimizer(learning_rate_pre).minimize(ae['
    cost'])
sess_pre = tf.Session()
sess_pre.run(tf.initialize_all_variables())

batch_size = 100
n_epochs = 10  ## NEED TO CHANGE THIS LATER LIKE 10
s_time = time.time()
for epoch_i in range(n_epochs):
    for batch_i in range(len(data.train.labels) // batch_size):  ##
    UPDATE A SAMPLING METHOD TO USE HERE
        batch_xs = random_batch(batch_size, edge_img) ## UPDATE A
    SAMPLING METHOD TO USE HERE
        train = np.array([img - mean_img for img in batch_xs])
        sess_pre.run(optimizer_pre, feed_dict={ae['x']: train})
    print(epoch_i, sess_pre.run(ae['cost'], feed_dict={ae['x']: train}))
e_time = time.time()

print("Time used for pre-training:", str(timedelta(seconds=int(round(
    e_time - s_time)))))


def new_conv_layer(input,              # The previous layer.
                   num_input_channels, # Num. channels in prev. layer.
                   filter_size,        # Width and height of each filter
    .
                   num_filters,        # Number of filters.
                   use_pooling=True,
                   layer=0):  # Use 2x2 max-pooling.

    # Shape of the filter-weights for the convolution.
    # This format is determined by the TensorFlow API.
    shape = [filter_size, filter_size, num_input_channels, num_filters]

    # Create new weights aka. filters with the given shape.
    weights = tf.Variable(sess_pre.run(ae['weights2d'])[layer])
```

```
265
266     # Create new biases, one for each filter.
267     biases = tf.Variable(sess_pre.run(ae['biases2d'])[layer])
268     layer = tf.nn.conv2d(input=input,
269                          filter=weights,
270                          strides=[1, 1, 1, 1],
271                          padding='SAME')
272
273     # Add the biases to the results of the convolution.
274     # A bias-value is added to each filter-channel.
275     layer += biases
276
277     # Use pooling to down-sample the image resolution?
278     if use_pooling:
279         layer = tf.nn.max_pool(value=layer,
280                                ksize=[1, 2, 2, 1],
281                                strides=[1, 2, 2, 1],
282                                padding='SAME')
283
284     layer = tf.nn.relu(layer)
285
286     return layer, weights
287
288
289 ### Helper-function for flattening a layer
290 def flatten_layer(layer):
291     # Get the shape of the input layer.
292     layer_shape = layer.get_shape()
293
294     num_features = layer_shape[1:4].num_elements()
295     layer_flat = tf.reshape(layer, [-1, num_features])
296
297     return layer_flat, num_features
298
299
300 ### Helper-function for creating a new Fully-Connected Layer
301
302 # This function creates a new fully-connected layer in the computational
      graph for TensorFlow. Nothing is actually calculated here, we are
    just adding the mathematical formulas to the TensorFlow graph.
303 #
304 # It is assumed that the input is a 2-dim tensor of shape `[num_images,
    num_inputs]`. The output is a 2-dim tensor of shape `[num_images,
    num_outputs]`.
305 def new_fc_layer(input,          # The previous layer.
306                  num_inputs,     # Num. inputs from prev. layer.
307                  num_outputs,    # Num. outputs.
308                  use_relu=True): # Use Rectified Linear Unit (ReLU)?
309
310     # Create new weights and biases.
311     weights = new_weights(shape=[num_inputs, num_outputs])
312     biases = new_biases(length=num_outputs)
313
314     # Calculate the layer as the matrix multiplication of
```

```
315     # the input and weights, and then add the bias-values.
316     layer = tf.matmul(input, weights) + biases
317 
318     # Use ReLU?
319     if use_relu:
320         layer = tf.nn.relu(layer)
321 
322     return layer
323 
324 
325 # ### Placeholder variables
326 x = tf.placeholder(tf.float32, shape=[None, img_size_flat], name='x')
327 x_image = tf.reshape(x, [-1, img_size, img_size, num_channels])
328 y_true = tf.placeholder(tf.float32, shape=[None, 10], name='y_true')
329 y_true_cls = tf.argmax(y_true, dimension=1)
330 
331 layer_conv1, weights_conv1 =     new_conv_layer(input=x_image,
332                    num_input_channels=num_channels,
333                    filter_size=filter_size1,
334                    num_filters=num_filters1,
335                    use_pooling=True,
336                    layer = 0)
337 
338 layer_conv2, weights_conv2 =     new_conv_layer(input=layer_conv1,
339                    num_input_channels=num_filters1,
340                    filter_size=filter_size2,
341                    num_filters=num_filters2,
342                    use_pooling=True,
343                    layer=1)
344 
345 layer_flat, num_features = flatten_layer(layer_conv2)
346 
347 # ### Fully-Connected Layer
348 #
349 # Add a fully-connected layer to the network. The input is the flattened
      layer from the previous convolution. The number of neurons or nodes
    in the fully-connected layer is `fc_size`. ReLU is used so we can
    learn non-linear relations.
350 layer_fc1 = new_fc_layer(input=layer_flat,
351                          num_inputs=num_features,
352                          num_outputs=fc_size,
353                          use_relu=True)
354 
355 # ### Fully-Connected Layer
356 #
357 # Add another fully-connected layer that outputs vectors of length 10
    for determining which of the 10 classes the input image belongs to.
    Note that ReLU is not used in this layer.
358 
359 layer_fc2 = new_fc_layer(input=layer_fc1,
360                          num_inputs=fc_size,
361                          num_outputs=num_classes,
362                          use_relu=False)
363 
```

```
y_pred = tf.nn.softmax(layer_fc2)

y_pred_cls = tf.argmax(y_pred, dimension=1)

cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits=layer_fc2
    ,
                                                        labels=y_true)
cost = tf.reduce_mean(cross_entropy)

optimizer = tf.train.AdamOptimizer(learning_rate=1e-4).minimize(cost)

correct_prediction = tf.equal(y_pred_cls, y_true_cls)

accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

session = tf.Session()

session.run(tf.initialize_all_variables())


### Helper-function to perform optimization iterations
train_batch_size = 64
total_iterations = 0

def optimize(num_iterations):
    # Ensure we update the global variable rather than a local copy.
    global total_iterations

    # Start-time used for printing time-usage below.
    start_time = time.time()

    for i in range(total_iterations,
                   total_iterations + num_iterations):

        # Get a batch of training examples.
        # x_batch now holds a batch of images and
        # y_true_batch are the true labels for those images.
        x_batch, y_true_batch = data.train.next_batch(train_batch_size)

        # Put the batch into a dict with the proper names
        # for placeholder variables in the TensorFlow graph.
        feed_dict_train = {x: x_batch,
                           y_true: y_true_batch}

        # Run the optimizer using this batch of training data.
        # TensorFlow assigns the variables in feed_dict_train
        # to the placeholder variables and then runs the optimizer.
        session.run(optimizer, feed_dict=feed_dict_train)

        # Print status every 100 iterations.
        if i % 100 == 0:
            # Calculate the accuracy on the training-set.
            acc = session.run(accuracy, feed_dict=feed_dict_train)

```

```
            # Message for printing.
            msg = "Optimization Iteration: {0:>6}, Training Accuracy:
    {1:>6.1%}"

            # Print it.
            print(msg.format(i + 1, acc))

    # Update the total number of iterations performed.
    total_iterations += num_iterations

    # Ending time.
    end_time = time.time()

    # Difference between start and end-times.
    time_dif = end_time - start_time

    # Print the time-usage.
    print("Time usage: " + str(timedelta(seconds=int(round(time_dif)))))


### Helper-function to plot example errors
def plot_example_errors(cls_pred, correct):
    incorrect = (correct == False)

    images = data.test.images[incorrect]

    # Get the predicted classes for those images.
    cls_pred = cls_pred[incorrect]

    # Get the true classes for those images.
    cls_true = data.test.cls[incorrect]

    # Plot the first 9 images.
    plot_images(images=images[0:9],
                cls_true=cls_true[0:9],
                cls_pred=cls_pred[0:9])

def plot_confusion_matrix(cls_pred):
    cls_true = data.test.cls

    # Get the confusion matrix using sklearn.
    cm = confusion_matrix(y_true=cls_true,
                          y_pred=cls_pred)

    # Print the confusion matrix as text.
    print(cm)

    # Plot the confusion matrix as an image.
    plt.matshow(cm)

    # Make various adjustments to the plot.
    plt.colorbar()
    tick_marks = np.arange(num_classes)
    plt.xticks(tick_marks, range(num_classes))
```

```
    plt.yticks(tick_marks, range(num_classes))
    plt.xlabel('Predicted')
    plt.ylabel('True')

    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()


### Helper-function for showing the performance

test_batch_size = 256

def print_test_accuracy(show_example_errors=False,
                        show_confusion_matrix=False):

    # Number of images in the test-set.
    num_test = len(data.test.images)

    # Allocate an array for the predicted classes which
    # will be calculated in batches and filled into this array.
    cls_pred = np.zeros(shape=num_test, dtype=np.int)

    i = 0

    while i < num_test:
        # The ending index for the next batch is denoted j.
        j = min(i + test_batch_size, num_test)

        # Get the images from the test-set between index i and j.
        images = data.test.images[i:j, :]

        # Get the associated labels.
        labels = data.test.labels[i:j, :]

        # Create a feed-dict with these images and labels.
        feed_dict = {x: images,
                     y_true: labels}

        # Calculate the predicted class using TensorFlow.
        cls_pred[i:j] = session.run(y_pred_cls, feed_dict=feed_dict)

        # Set the start-index for the next batch to the
        # end-index of the current batch.
        i = j

    # Convenience variable for the true class-numbers of the test-set.
    cls_true = data.test.cls

    # Create a boolean array whether each image is correctly classified.
    correct = (cls_true == cls_pred)

    # Calculate the number of correctly classified images.
    # When summing a boolean array, False means 0 and True means 1.
```

```
    correct_sum = correct.sum()

    # Classification accuracy is the number of correctly classified
    # images divided by the total number of images in the test-set.
    acc = float(correct_sum) / num_test

    # Print the accuracy.
    msg = "Accuracy on Test-Set: {0:.1%} ({1} / {2})"
    print(msg.format(acc, correct_sum, num_test))

    # Plot some examples of mis-classifications, if desired.
    if show_example_errors:
        print("Example errors:")
        plot_example_errors(cls_pred=cls_pred, correct=correct)

    # Plot the confusion matrix, if desired.
    if show_confusion_matrix:
        print("Confusion Matrix:")
        plot_confusion_matrix(cls_pred=cls_pred)


## Performance before any optimization
print_test_accuracy()

## Performance after 1 optimization iteration
optimize(num_iterations=1)
print_test_accuracy()

## Performance after 100 optimization iterations
optimize(num_iterations=99) # We already performed 1 iteration above.
print_test_accuracy(show_example_errors=True)

optimize(num_iterations=900) # We performed 100 iterations above.
print_test_accuracy(show_example_errors=True)
optimize(num_iterations=9000) # We performed 1000 iterations above.
print_test_accuracy(show_example_errors=True,
                    show_confusion_matrix=True)
```

Listing A.1: Pretraining with autoencoder and convolutional neural netowork on MNIST Datasets

A.3 Pretraining with anistropic Diffusion images and Convolutional Neural Network

```
#########################################################################
# Author: Xiaotian Zhao
# The following program is modified based on code '06_CIFAR-10.ipynb'
# from https://github.com/Hvass-Labs/TensorFlow-Tutorials.git
#########################################################################
import sys
sys.path.append('/usr/lib/python3/dist-packages')## Imports
```

```
8
9  import matplotlib.pyplot as plt
10 import tensorflow as tf
11 import numpy as np
12 from sklearn.metrics import confusion_matrix
13 from skimage import feature
14 from skimage.transform import rotate
15 from PIL import Image
16 import time
17 from datetime import timedelta
18 import math
19 import os
20 import gc
21 import scipy.io as sio
22
23 def plot_images(images, cls_true, cls_pred=None, smooth=True):
24     assert len(images) == len(cls_true) ==9
25
26     fig, axes = plt.subplots(3,3)
27
28     if cls_pred is None:
29         hspace = 0.3
30     else:
31         hspace = 0.6
32
33     fig.subplots_adjust(hspace=hspace, wspace=0.3)
34
35     for i, ax in enumerate(axes.flat):
36         if smooth:
37             interpolation = 'spline16'
38         else:
39             interpolation = 'nearest'
40
41         ax.imshow(images[i,:,:,:],
42                   interpolation = interpolation)
43
44         cls_true_name = class_names[cls_true[i]]
45
46         if cls_pred is None:
47             xlabel = "True: {0}".format(cls_true_name)
48         else:
49             cls_pred_name = class_names[cls_pred[i]]
50             xlabel = "True: {0}\nPred: {1}".format(cls_true_name,
    cls_pred_name)
51         ax.set_xlabel(xlabel)
52
53         ax.set_xticks([])
54         ax.set_yticks([])
55
56     plt.show()
57
58 def plot_example_errors(cls_pred, correct):
59     incorrect = (correct == False)
60
```

```
    # Get the images from the test-set that have been
    # incorrectly classified.
    images = images_test[incorrect]

    # Get the predicted classes for those images.
    cls_pred = cls_pred[incorrect]

    # Get the true classes for those images.
    cls_true = cls_test[incorrect]

    # Plot the first 9 images.
    plot_images(images=images[0:9],
                cls_true=cls_true[0:9],
                cls_pred=cls_pred[0:9])


def plot_confusion_matrix(cls_pred):
    cm = confusion_matrix(y_true=cls_test,  # True class for test-set.
                          y_pred=cls_pred)  # Predicted class.

    # Print the confusion matrix as text.
    for i in range(num_classes):
        # Append the class-name to each line.
        class_name = "({}) {}".format(i, class_names[i])
        print(cm[i, :], class_name)

    # Print the class-numbers for easy reference.
    class_numbers = [" ({0})".format(i) for i in range(num_classes)]
    print("".join(class_numbers))


# In[6]:

def plot_conv_weights(weights, input_channel=0):
    w = session.run(weights)

    # Print statistics for the weights.
    print("Min:  {0:.5f}, Max:   {1:.5f}".format(w.min(), w.max()))
    print("Mean: {0:.5f}, Stdev: {1:.5f}".format(w.mean(), w.std()))
    w_min = np.min(w)
    w_max = np.max(w)
    abs_max = max(abs(w_min), abs(w_max))

    # Number of filters used in the conv. layer.
    num_filters = w.shape[3]
    num_grids = int(math.ceil(math.sqrt(num_filters)))

    # Create figure with a grid of sub-plots.
    fig, axes = plt.subplots(num_grids, num_grids)

    # Plot all the filter-weights.
    for i, ax in enumerate(axes.flat):
        # Only plot the valid filter-weights.
        if i<num_filters:
```

```
            img = w[:, :, input_channel, i]

            # Plot image.
            ax.imshow(img, vmin=-abs_max, vmax=abs_max,
                      interpolation='nearest', cmap='seismic')

        # Remove ticks from the plot.
        ax.set_xticks([])
        ax.set_yticks([])

    plt.show()

def plot_image(image):
    # Create figure with sub-plots.
    fig, axes = plt.subplots(1, 2)

    # References to the sub-plots.
    ax0 = axes.flat[0]
    ax1 = axes.flat[1]

    # Show raw and smoothened images in sub-plots.
    ax0.imshow(image, interpolation='nearest')
    ax1.imshow(image, interpolation='spline16')

    # Set labels.
    ax0.set_xlabel('Raw')
    ax1.set_xlabel('Smooth')

    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()

import cifar10 ## changes to made in download.py
cifar10.data_path = "data/CIFAR-10/"
file_path = cifar10._get_file_path()

class_names = cifar10.load_class_names()
images_train, cls_train, labels_train = cifar10.load_training_data()
images_test, cls_test, labels_test = cifar10.load_test_data()
print("Size of:")
print ("- Training-set: \t\t{}".format(len(labels_train)))
print ("- Test-set:     \t\t{}".format(len(labels_test)))
def pre_process_image(image, training):
    if training:
        image = tf.random_crop(image, size=[img_size_cropped,
   img_size_cropped, num_channels])
        image = tf.image.random_flip_left_right(image)
        image = tf.image.random_hue(image, max_delta=0.05)
        image = tf.image.random_contrast(image,lower=0.3, upper = 1.0)
        image = tf.image.random_brightness(image, max_delta=0.2)
        image = tf.image.random_saturation(image, lower=0.0, upper=2.0)

        ## we limit the range of the value of image because some of the
   image\
```

```
        ## is overflowed or underflowed
        image = tf.minimum(image, 1.0)
        image = tf.maximum(image, 0.0)

    else:
        #image = tf.image.rot90(image,k=1) #### WE ROTATE HERE AS WELL BECAUSE THIS IS TEST IMAGES
        image = tf.image.resize_image_with_crop_or_pad(image,
                                                       target_height=img_size_cropped,
                                                       target_width = img_size_cropped)

    return image

#### Load a Anistropic Diffusion image
# Number of images for each batch-file in the training-set.
_images_per_file = 10000
_num_files_train = 5
# Total number of images in the training-set.
# This is used to pre-allocate arrays for efficiency.
_num_images_train = _num_files_train * _images_per_file

inputData =sio.loadmat('data/CIFAR-10/allCifar10.mat')

Adimages_train = np.zeros(shape=[_num_images_train, 32, 32, 3], dtype=float)

# for each batch files
# for i in range(5):
num_images = 10000

for i in range(10000):
    Adimages_train[i,:,:, :] = rotate(inputData["T_batch1"][:,:,:,i],270)

    Adimages_train[i+10000,:,:, :] = rotate(inputData["T_batch2"][:,:,:,i],270)
    Adimages_train[i+20000,:,:, :] = rotate(inputData["T_batch3"][:,:,:,i],270)
    Adimages_train[i+30000,:,:, :] = rotate(inputData["T_batch4"][:,:,:,i],270)
    Adimages_train[i+40000,:,:, :] = rotate(inputData["T_batch5"][:,:,:,i],270)

from cifar10 import img_size, num_channels, num_classes
num_channels = 3 # assume that the color image is 3 channels
img_size_cropped = 24

images = Adimages_train[0:9]

cls_true = cls_test[0:9]

plot_images(images=images, cls_true=cls_true, smooth=False)
```

```
212
213 # WITH SMOOTHING
214 plot_images(images=images, cls_true=cls_true, smooth=True)
215
216 x = tf.placeholder(tf.float32, shape=[None, img_size, img_size,
    num_channels], name='x')
217 x_image = tf.reshape(x, [-1, img_size, img_size, num_channels])
218 y_true = tf.placeholder(tf.float32, shape=[None, num_classes],name='
    y_true')
219 y_true_cls = tf.argmax(y_true, dimension=1)
220
221
222 ## preprocessing image
223
224 def pre_process(images, training):
225     images = tf.map_fn(lambda image: pre_process_image(image, training),
    images)
226
227     return images
228
229 distorted_images = pre_process(images=x, training=True)
230
231 #### Network Structure variables
232 filter_size1 = 5
233 num_filters1 = 64
234 filter_size2 = 5
235 num_filters2 = 64
236 fc_size = 128
237
238 def new_weights(shape):
239     return tf.Variable(tf.truncated_normal(shape, stddev=0.05))
240
241 def new_biases(length):
242     return tf.Variable(tf.constant(0.05,shape=[length]))
243
244
245 ### Pretraining with stacked autoencoder
246
247 def autoencoder(input_shape=[None, img_size, img_size, num_channels],
248                 n_filters=[3, 64, 64],
249                 filter_sizes=[5, 5],
250                 corruption=False):
251     # input to the network
252     x = tf.placeholder(
253         tf.float32, input_shape, name='x')
254
255     # ensure 2-d is converted to square tensor.
256     if len(x.get_shape()) == 2:
257         x_dim = np.sqrt(x.get_shape().as_list()[1])
258         if x_dim != int(x_dim):
259             raise ValueError('Unsupported input dimensions')
260         x_dim = int(x_dim)
261         x_tensor = tf.reshape(
262             x, [-1, x_dim, x_dim, n_filters[0]])
```

```
263     elif len(x.get_shape()) == 4:
264         x_tensor = x
265     else:
266         raise ValueError('Unsupported input dimensions')
267     current_input = x_tensor
268 
269     # %%
270     # Optionally apply denoising autoencoder
271 #     if corruption:
272 #         current_input = corrupt(current_input)
273 
274     # %%
275     # Build the encoder
276     encoder = []
277     shapes = []
278     biases = []
279     for layer_i, n_output in enumerate(n_filters[1:]):
280         n_input = current_input.get_shape().as_list()[3]
281         shapes.append(current_input.get_shape().as_list())
282         W = tf.Variable(
283             tf.random_uniform([
284                 filter_sizes[layer_i],
285                 filter_sizes[layer_i],
286                 n_input, n_output],
287                 -1.0 / math.sqrt(n_input),
288                 1.0 / math.sqrt(n_input)))
289         b = tf.Variable(tf.zeros([n_output]))
290         biases.append(b)
291         encoder.append(W)
292         output = tf.nn.relu(
293             tf.add(tf.nn.conv2d(
294                 current_input, W, strides=[1, 2, 2, 1], padding='SAME'),
    b))
295         current_input = output
296 
297     # %%
298     # store the latent representation
299     z = current_input
300     encoder.reverse() # because we want to deal with inner deep layer
    than outer layer and so on.
301     biases.reverse()
302     shapes.reverse()
303 
304     # %%
305     # Build the decoder using the same weights
306     for layer_i, shape in enumerate(shapes):
307         W = encoder[layer_i]
308         b = tf.Variable(tf.zeros([W.get_shape().as_list()[2]]))
309         output = tf.nn.relu(tf.add(
310             tf.nn.conv2d_transpose(
311                 current_input, W,
312                 tf.pack([tf.shape(x)[0], shape[1], shape[2], shape[3]]),
313                 strides=[1, 2, 2, 1], padding='SAME'), b ) )
314 
```

```
        current_input = output

    # %%
    # now have the reconstruction through the network
    y = current_input
    # cost function measures pixel-wise difference
    cost = tf.reduce_sum(tf.square(y - x_tensor))

    # %%
    return {'x': x, 'z': z, 'y': y, 'cost': cost, 'weights2d':encoder,
            'biases2d':biases}


# In[29]:

## training data generator
def random_batch_pre(train_batch_size):
    num_images = len(Adimages_train)
    idx = np.random.choice(num_images,
                           size=train_batch_size,
                           replace=False)
    # Use the random index to select random images and labels.
    x_batch = Adimages_train[idx, :, :, :]
    y_batch = labels_train[idx, :]

    return x_batch, y_batch


# In[30]:

ae = autoencoder()
mean_img = np.mean(Adimages_train, axis=0)
learning_rate_pre = 0.01
optimizer_pre = tf.train.AdamOptimizer(learning_rate_pre).minimize(ae['
    cost'])
sess_pre = tf.Session()
sess_pre.run(tf.initialize_all_variables())

batch_size = 10 ### This number can not be large like more than 50 or so
    , it will worsen the result
n_epochs = 100 ## NEED TO CHANGE THIS LATER LIKE 10
s_time = time.time()
for epoch_i in range(n_epochs):
    for batch_i in range(len(images_train) // batch_size): ## UPDATE A
    SAMPLING METHOD TO USE HERE
        batch_xs, _ = random_batch_pre(batch_size) ## UPDATE A SAMPLING
    METHOD TO USE HERE
        train = np.array([img - mean_img for img in batch_xs])
        sess_pre.run(optimizer_pre, feed_dict={ae['x']: train})
    print(epoch_i, sess_pre.run(ae['cost'], feed_dict={ae['x']: train}))

e_time = time.time()

```

```
364 print("Time used for pre-training:", str(timedelta(seconds=int(round(
    e_time - s_time)))))
365
366 def new_conv_layer(input,
367                    num_input_channels,
368                    filter_size,
369                    num_filters,
370                    use_pooling=True,
371                    layer=0):
372     shape = [filter_size, filter_size, num_input_channels, num_filters]
373
374     weights = sess_pre.run(ae['weights2d'])[layer]
375
376     biases = sess_pre.run(ae['biases2d'])[layer]
377
378     layer = tf.nn.conv2d(input=input, filter=weights, strides=[1,1,1,1],
379                          padding='SAME')
380
381     layer += biases
382
383     if use_pooling:
384         layer = tf.nn.max_pool(value=layer,
385                                ksize=[1,2,2,1],
386                                strides=[1,2,2,1],
387                                padding='SAME')
388     layer = tf.nn.relu(layer)
389
390     return layer, weights
391
392
393 # #### Flatten and fullyconnected layer
394
395 def flatten_layer(layer):
396     layer_shape = layer.get_shape()
397     num_features = layer_shape[1:4].num_elements()
398     layer_flat = tf.reshape(layer,[-1,num_features])
399     return layer_flat, num_features
400
401 def new_fc_layer(input,
402                  num_inputs,
403                  num_outputs,
404                  use_relu=True):
405     weights = new_weights(shape=[num_inputs, num_outputs])
406     biases = new_biases(length=num_outputs)
407     layer=tf.matmul(input,weights) + biases
408     if use_relu:
409         layer = tf.nn.relu(layer)
410     return layer
411
412
413 # #### main network
414 # first convolution layer
415 layer_conv1, weights_conv1 =      new_conv_layer(input=pre_process(images
    = x, training = True),
```

```
                    num_input_channels= num_channels,
                    filter_size=filter_size1,
                    num_filters=num_filters1,
                    use_pooling=True,
                    layer = 1)

# second convolution layer
layer_conv2, weights_conv2 =     new_conv_layer(input=layer_conv1,
                    num_input_channels=num_filters1,
                    filter_size=filter_size2,
                    num_filters=num_filters2,
                    use_pooling=True,
                    layer = 0)
# flatten Layer
layer_flat, num_features = flatten_layer(layer_conv2)

# fully connected layer1
layer_fc1 = new_fc_layer(input=layer_flat,
                         num_inputs=num_features,
                         num_outputs = fc_size,
                         use_relu=True)

# fully connected layer2
layer_fc2 = new_fc_layer(input=layer_fc1,
                         num_inputs=fc_size,
                         num_outputs=num_classes,
                         use_relu=False)


cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits=layer_fc2
    ,
                                                        labels=y_true)
cost = tf.reduce_mean(cross_entropy)

# In[34]:

global_step = tf.Variable(initial_value=0,
                          name='global_step', trainable=False)
# In[35]:

optimizer = tf.train.AdamOptimizer(learning_rate=1e-5).minimize(cost,
    global_step=global_step)


## predicted class
y_pred = tf.nn.softmax(layer_fc2)
y_pred_cls = tf.argmax(y_pred, dimension=1)
correct_prediction = tf.equal(y_pred_cls, y_true_cls)
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

## Saver
saver = tf.train.Saver()
session = tf.Session()
save_dir = 'checkpoints2/'
```

```
# create the directory if it dose not exist
if not os.path.exists(save_dir):
    os.makedirs(save_dir)

save_path = save_dir + 'cifar10_cnn'
try:
    print("Trying to restore last checkpoint")
    last_chk_path = tf.train.latest_checkpoint(checkpoint_dir=save_dir)
    # use tensorflow function to find last checkpoint

    saver.restore(session=session, save_path= last_chk_path)
    print("Restored checkpoint from:", last_chk_path)

except:
    # If the above failed for some reason, simply
    # initialize all the variables for the TensorFlow graph.
    print("Failed to restore checkpoint. Initializing variables instead.
    ")
    session.run(tf.initialize_all_variables())

train_batch_size = 64
######################
# Creating Batch for Main Training
# Using *images_train**
######################
def random_batch():
    num_images = len(images_train)
    idx = np.random.choice(num_images,
                           size=train_batch_size,
                           replace=False)
    # Use the random index to select random images and labels.
    x_batch = images_train[idx, :, :, :]
    y_batch = labels_train[idx, :]

    return x_batch, y_batch

# ### Helper function to do optimization
def optimize(num_iterations):
    # Start-time used for printing time-usage below.
    start_time = time.time()

    for i in range(num_iterations):
        x_batch, y_true_batch = random_batch()
        feed_dict_train = {x: x_batch,
                           y_true: y_true_batch}
        i_global, _ = session.run([global_step, optimizer],
                                  feed_dict=feed_dict_train)

        # Print status to screen every 100 iterations (and last).
        if (i_global % 100 == 0) or (i == num_iterations - 1):
            # Calculate the accuracy on the training-batch.
            batch_acc = session.run(accuracy,
                                    feed_dict=feed_dict_train)

```

```
            # Print status.
            msg = "Global Step: {0:>6}, Training Batch Accuracy:
    {1:>6.1%}"
            print(msg.format(i_global, batch_acc))

        # Save a checkpoint to disk every 1000 iterations (and last).
        if (i_global % 1000 == 0) or (i == num_iterations - 1):
            saver.save(session,
                       save_path=save_path,
                       global_step=global_step)

            print("Saved checkpoint.")

    # Ending time.
    end_time = time.time()

    # Difference between start and end-times.
    time_dif = end_time - start_time

    # Print the time-usage.
    print("Time usage: " + str(timedelta(seconds=int(round(time_dif)))))


batch_size= 256
def predict_cls(images, labels, cls_true):
    num_images = len(images)
    cls_pred = np.zeros(shape=num_images, dtype=np.int)

    i=0
    while i < num_images:
        # The ending index for the next batch is denoted j.
        j = min(i + batch_size, num_images)

        feed_dict = {x: images[i:j, :],
                     y_true: labels[i:j, :]}

        # Calculate the predicted class using TensorFlow.
        cls_pred[i:j] = session.run(y_pred_cls, feed_dict=feed_dict)

        i = j

    # Create a boolean array whether each image is correctly classified.
    correct = (cls_true == cls_pred)

    return correct, cls_pred

def predict_cls_test():
    return predict_cls(images = images_test,
                       labels = labels_test,
                       cls_true = cls_test)
def classification_accuracy(correct):
    return correct.mean(), correct.sum()


```

```
574 def print_test_accuracy(show_example_errors=False,
575                          show_confusion_matrix=False):
576     correct, cls_pred = predict_cls_test()
577 
578     # Classification accuracy and the number of correct classifications.
579     acc, num_correct = classification_accuracy(correct)
580 
581     # Number of images being classified.
582     num_images = len(correct)
583 
584     # Print the accuracy.
585     msg = "Accuracy on Test-Set: {0:.1%} ({1} / {2})"
586     print(msg.format(acc, num_correct, num_images))
587 
588     # Plot some examples of mis-classifications, if desired.
589     if show_example_errors:
590         print("Example errors:")
591         plot_example_errors(cls_pred=cls_pred, correct=correct)
592 
593     # Plot the confusion matrix, if desired.
594     if show_confusion_matrix:
595         print("Confusion Matrix:")
596         plot_confusion_matrix(cls_pred=cls_pred)
597 
598 
599 # ### plot example distoreted images
600 def plot_distorted_image(image, cls_true):
601     # Repeat the input image 9 times.
602     image_duplicates = np.repeat(image[np.newaxis, :, :, :], 9, axis=0)
603 
604     # Create a feed-dict for TensorFlow.
605     feed_dict = {x: image_duplicates}
606 
607     # Calculate only the pre-processing of the TensorFlow graph
608     # which distorts the images in the feed-dict.
609     result = session.run(distorted_images, feed_dict=feed_dict)
610 
611     # Plot the images.
612     plot_images(images=result, cls_true=np.repeat(cls_true, 9))
613 
614 def get_test_image(i):
615     return images_test[i, :, :, :], cls_test[i]
616 
617 img, cls = get_test_image(16)
618 plot_distorted_image(img, cls)
619 
620 optimize(num_iterations=100000)
621 
622 print_test_accuracy(show_example_errors=True,
623                     show_confusion_matrix=True)
624 summary_writer = tf.train.SummaryWriter('cifar10_exec_data', graph_def=
    session.graph_def)
```

Listing A.2: pretraining with anistropic Diffusion images

A.4 Convolutional Neural Network with CIFAR-10 images

```
1
2 ########################################################################
3 # # Author: Xiaotian Zhao
4 # The following program is modified based on code '08_Transfer_Learning.
    ipynb'
5 # from https://github.com/Hvass-Labs/TensorFlow-Tutorials.git
6 ########################################################################
7 import matplotlib.pyplot as plt
8 import prettytensor as pt
9 import tensorflow as tf
10 import numpy as np
11 import time
12 from datetime import timedelta
13 import os
14 import pip
15 # Functions and classes for loading and using the Inception model.
16 import inception
17 from sklearn.metrics import confusion_matrix
18 import math
19 import os
20 import scipy.io as sio
21
22 # We use Pretty Tensor to define the new classifier.
23
24 tf.__version__
25
26
27 # ## Load Data for Oxford Flower
28
29 import OxfordFlower
30
31
32 # The data dimensions have already been defined in the cifar10 module,
    so we just need to import the ones we need.
33
34 from OxfordFlower import img_size, num_channels, num_classes
35 num_channels = 3
36
37
38 # Set the path for storing the data-set on your computer.
39 OxfordFlower.data_path = "data/Flowers/"
40
41
42 images_train, cls_train, labels_train = OxfordFlower.getTrainingSets(
    choice = 1)
43
44
45 images_test, cls_test, labels_test = OxfordFlower.getTestSets(choice =
    1)
46
```

```
47
48
49 print("Size of:")
50 print("- Training-set:\t\t{}".format(len(images_train)))
51 print("- Test-set:\t\t{}".format(len(images_test)))
52
53
54 def plot_images(images, cls_true, cls_pred=None, smooth=True):
55
56     assert len(images) == len(cls_true)
57
58     # Create figure with sub-plots.
59     fig, axes = plt.subplots(3, 3)
60
61     # Adjust vertical spacing.
62     if cls_pred is None:
63         hspace = 0.3
64     else:
65         hspace = 0.6
66     fig.subplots_adjust(hspace=hspace, wspace=0.3)
67
68     # Interpolation type.
69     if smooth:
70         interpolation = 'spline16'
71     else:
72         interpolation = 'nearest'
73
74     for i, ax in enumerate(axes.flat):
75         # There may be less than 9 images, ensure it doesn't crash.
76         if i < len(images):
77             # Plot image.
78             ax.imshow(images[i],
79                       interpolation=interpolation)
80
81             # Name of the true class.
82             xlabel = cls_true[i]
83 #             cls_true_name = class_names[cls_true[i]]
84
85 #             # Show true and predicted classes.
86 #             if cls_pred is None:
87 #                 xlabel = "True: {0}".format(cls_true_name)
88 #             else:
89 #                 # Name of the predicted class.
90 #                 cls_pred_name = class_names[cls_pred[i]]
91
92 #                 xlabel = "True: {0}\nPred: {1}".format(cls_true_name,
    cls_pred_name)
93
94             # Show the classes as the label on the x-axis.
95             ax.set_xlabel(xlabel)
96
97         # Remove ticks from the plot.
98         ax.set_xticks([])
99         ax.set_yticks([])
```

```
100
101     # Ensure the plot is shown correctly with multiple plots
102     # in a single Notebook cell.
103     plt.show()
104
105
106 # ### Plot a few images to see if data is correct
107
108 # In[14]:
109
110 # Get the first images from the test-set.
111 images = images_test[0:9]
112
113 # Get the true classes for those images.
114 cls_true = cls_test[0:9]
115
116 # Plot the images and labels using our helper-function above.
117 plot_images(images=images, cls_true=cls_true, smooth=False)
118
119
120 model = inception.Inception()
121
122
123
124 from inception import transfer_values_cache
125
126
127 file_path_cache_train = os.path.join(OxfordFlower.data_path, '
    inception_Flowers1.1_train.pkl')
128 file_path_cache_test = os.path.join(OxfordFlower.data_path, '
    inception_Flowers1.1_test.pkl')
129
130
131 # In[20]:
132
133 print("Processing Inception transfer-values for training-images ...")
134
135 # Scale images because Inception needs pixels to be between 0 and 255,
136 # while the CIFAR-10 functions return pixels between 0.0 and 1.0
137 images_scaled = images_train * 255.0
138
139 # If transfer-values have already been calculated then reload them,
140 # otherwise calculate them and save them to a cache-file.
141 transfer_values_train = transfer_values_cache(cache_path=
    file_path_cache_train,
142                                                images=images_scaled,
143                                                model=model)
144
145
146 # In[21]:
147
148 print("Processing Inception transfer-values for test-images ...")
149
150 # Scale images because Inception needs pixels to be between 0 and 255,
```

```
# while the CIFAR-10 functions return pixels between 0.0 and 1.0
images_scaled = images_test * 255.0

# If transfer-values have already been calculated then reload them,
# otherwise calculate them and save them to a cache-file.
transfer_values_test = transfer_values_cache(cache_path=
    file_path_cache_test,
                                             images=images_scaled,
                                             model=model)


transfer_values_train.shape


# Similarly, there are 10,000 images in the test-set with 2048 transfer-
    values for each image.

# In[24]:

transfer_values_test.shape


### Helper-function for plotting transfer-values

# In[25]:

def plot_transfer_values(i):
    print("Input image:")

    # Plot the i'th image from the test-set.
    plt.imshow(images_test[i], interpolation='nearest')
    plt.show()

    print("Transfer-values for the image using Inception model:")

    # Transform the transfer-values into an image.
    img = transfer_values_test[i]
    img = img.reshape((32, 64))

    # Plot the image for the transfer-values.
    plt.imshow(img, interpolation='nearest', cmap='Reds')
    plt.show()


# In[26]:

plot_transfer_values(i=16)


# In[27]:

plot_transfer_values(i=17)


```

```
# ## Analysis of Transfer-Values using PCA

# Use Principal Component Analysis (PCA) from scikit-learn to reduce the
    array-lengths of the transfer-values from 2048 to 2 so they can be
    plotted.

# In[28]:

from sklearn.decomposition import PCA


# Create a new PCA-object and set the target array-length to 2.

# In[29]:

pca = PCA(n_components=2)


# It takes a while to compute the PCA so the number of samples has been
    limited to 3000. You can try and use the full training-set if you
    like.

# In[30]:

transfer_values = transfer_values_train[0:3000]


# Get the class-numbers for the samples you selected.

# In[31]:

cls = cls_train[0:3000]


# Check that the array has 3000 samples and 2048 transfer-values for
    each sample.

# In[32]:

transfer_values.shape


# Use PCA to reduce the transfer-value arrays from 2048 to 2 elements.

# In[33]:

transfer_values_reduced = pca.fit_transform(transfer_values)


# Check that it is now an array with 3000 samples and 2 values per
    sample.

# In[34]:

```

```
transfer_values_reduced.shape


# Helper-function for plotting the reduced transfer-values.

# In[35]:

def plot_scatter(values, cls):
    # Create a color-map with a different color for each class.
    import matplotlib.cm as cm
    cmap = cm.rainbow(np.linspace(0.0, 1.0, num_classes))

    # Get the color for each sample.
    colors = cmap[cls]

    # Extract the x- and y-values.
    x = values[:, 0]
    y = values[:, 1]

    # Plot it.
    plt.scatter(x, y, color=colors)
    plt.show()

plot_scatter(transfer_values_reduced, cls)


## Analysis of Transfer-Values using t-SNE

# In[37]:

from sklearn.manifold import TSNE


# Another method for doing dimensionality reduction is t-SNE.
    Unfortunately, t-SNE is very slow so we first use PCA to reduce the
    transfer-values from 2048 to 50 elements.

# In[38]:

pca = PCA(n_components=50)
transfer_values_50d = pca.fit_transform(transfer_values)


# Create a new t-SNE object for the final dimensionality reduction and
    set the target to 2-dim.

# In[39]:

tsne = TSNE(n_components=2)


# Perform the final reduction using t-SNE. The current implemenation of
    t-SNE in scikit-learn cannot handle data with many samples so this
    might crash if you use the full training-set.
```

```
300
301 # In[40]:
302
303 transfer_values_reduced = tsne.fit_transform(transfer_values_50d)
304
305
306 # Check that it is now an array with 3000 samples and 2 transfer-values
    per sample.
307
308 # In[41]:
309
310 transfer_values_reduced.shape
311
312
313 plot_scatter(transfer_values_reduced, cls)
314
315
316
317 transfer_len = model.transfer_len
318
319
320 x = tf.placeholder(tf.float32, shape=[None, transfer_len], name='x')
321
322
323 y_true = tf.placeholder(tf.float32, shape=[None, num_classes], name='
    y_true')
324
325
326 y_true_cls = tf.argmax(y_true, dimension=1)
327
328
329 ### Neural Network
330
331 # Wrap the transfer-values as a Pretty Tensor object.
332 x_pretty = pt.wrap(x)
333
334 with pt.defaults_scope(activation_fn=tf.nn.relu):
335     y_pred, loss = x_pretty.fully_connected(size=1000, name='layer_fc1')
    .softmax_classifier(num_classes, labels=y_true)
336
337
338 ### Optimization Method
339
340
341 global_step = tf.Variable(initial_value=0,
342                           name='global_step', trainable=False)
343
344
345
346 optimizer = tf.train.AdamOptimizer(learning_rate=1e-4).minimize(loss,
    global_step)
347
348
349 y_pred_cls = tf.argmax(y_pred, dimension=1)
```

```
350
351
352 correct_prediction = tf.equal(y_pred_cls, y_true_cls)
353
354
355 # The classification accuracy is calculated by first type-casting the
        array of booleans to floats, so that False becomes 0 and True becomes
         1, and then taking the average of these numbers.
356
357 # In[31]:
358
359 accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
360
361
362 # ## TensorFlow Run
363
364 session = tf.Session()
365
366 # ### Initialize Variables
367
368 session.run(tf.initialize_all_variables())
369
370
371 # ### Helper-function to get a random training-batch
372
373
374 # In[34]:
375
376 train_batch_size = 64
377
378
379 # Function for selecting a random batch of transfer-values from the
        training-set.
380
381 # In[35]:
382
383 def random_batch():
384     # Number of images (transfer-values) in the training-set.
385     num_images = len(transfer_values_train)
386
387     # Create a random index.
388     idx = np.random.choice(num_images,
389                            size=train_batch_size,
390                            replace=False)
391
392     # Use the random index to select random x and y-values.
393     # We use the transfer-values instead of images as x-values.
394     x_batch = transfer_values_train[idx]
395     y_batch = labels_train[idx]
396
397     return x_batch, y_batch
398
399
400 # ### Helper-function to perform optimization
```

```
401
402 def optimize(num_iterations):
403     # Start-time used for printing time-usage below.
404     start_time = time.time()
405
406     for i in range(num_iterations):
407         # Get a batch of training examples.
408         # x_batch now holds a batch of images (transfer-values) and
409         # y_true_batch are the true labels for those images.
410         x_batch, y_true_batch = random_batch()
411
412         # Put the batch into a dict with the proper names
413         # for placeholder variables in the TensorFlow graph.
414         feed_dict_train = {x: x_batch,
415                            y_true: y_true_batch}
416
417         # Run the optimizer using this batch of training data.
418         # TensorFlow assigns the variables in feed_dict_train
419         # to the placeholder variables and then runs the optimizer.
420         # We also want to retrieve the global_step counter.
421         i_global, _ = session.run([global_step, optimizer],
422                                   feed_dict=feed_dict_train)
423
424         # Print status to screen every 100 iterations (and last).
425         if (i_global % 100 == 0) or (i == num_iterations - 1):
426             # Calculate the accuracy on the training-batch.
427             batch_acc = session.run(accuracy,
428                                     feed_dict=feed_dict_train)
429
430             # Print status.
431             msg = "Global Step: {0:>6}, Training Batch Accuracy:
    {1:>6.1%}"
432             print(msg.format(i_global, batch_acc))
433
434     # Ending time.
435     end_time = time.time()
436
437     # Difference between start and end-times.
438     time_dif = end_time - start_time
439
440     # Print the time-usage.
441     print("Time usage: " + str(timedelta(seconds=int(round(time_dif)))))
442
443
444 # In[37]:
445
446 def plot_example_errors(cls_pred, correct):
447
448     incorrect = (correct == False)
449
450     # Get the images from the test-set that have been
451     # incorrectly classified.
452     images = images_test[incorrect]
453
```

```
    # Get the predicted classes for those images.
    cls_pred = cls_pred[incorrect]

    # Get the true classes for those images.
    cls_true = cls_test[incorrect]

    n = min(9, len(images))

    # Plot the first n images.
    plot_images(images=images[0:n],
                cls_true=cls_true[0:n],
                cls_pred=cls_pred[0:n])



# Import a function from sklearn to calculate the confusion-matrix.
from sklearn.metrics import confusion_matrix

def plot_confusion_matrix(cls_pred):
    # This is called from print_test_accuracy() below.

    # cls_pred is an array of the predicted class-number for
    # all images in the test-set.

    # Get the confusion matrix using sklearn.
    cm = confusion_matrix(y_true=cls_test,  # True class for test-set.
                          y_pred=cls_pred)  # Predicted class.

    # Print the confusion matrix as text.
    for i in range(num_classes):
        # Append the class-name to each line.
        class_name = "({}) {}".format(i, class_names[i])
        print(cm[i, :], class_name)

    # Print the class-numbers for easy reference.
    class_numbers = [" ({0})".format(i) for i in range(num_classes)]
    print("".join(class_numbers))




# Split the data-set in batches of this size to limit RAM usage.
batch_size = 256

def predict_cls(transfer_values, labels, cls_true):
    # Number of images.
    num_images = len(transfer_values)

    # Allocate an array for the predicted classes which
    # will be calculated in batches and filled into this array.
    cls_pred = np.zeros(shape=num_images, dtype=np.int)

    # Now calculate the predicted classes for the batches.
    # We will just iterate through all the batches.
```

```
    # There might be a more clever and Pythonic way of doing this.

    # The starting index for the next batch is denoted i.
    i = 0

    while i < num_images:
        # The ending index for the next batch is denoted j.
        j = min(i + batch_size, num_images)

        # Create a feed-dict with the images and labels
        # between index i and j.
        feed_dict = {x: transfer_values[i:j],
                     y_true: labels[i:j]}

        # Calculate the predicted class using TensorFlow.
        cls_pred[i:j] = session.run(y_pred_cls, feed_dict=feed_dict)

        # Set the start-index for the next batch to the
        # end-index of the current batch.
        i = j
    print('shape of cls_ture', cls_true.shape)
    print('shape of cls_pred', cls_pred.shape)
    # Create a boolean array whether each image is correctly classified.
    correct = (cls_true == cls_pred)

    return correct, cls_pred


# Calculate the predicted class for the test-set.

# In[40]:

def predict_cls_test():
    return predict_cls(transfer_values = transfer_values_test,
                       labels = labels_test,
                       cls_true = cls_test)

# In[41]:

def classification_accuracy(correct):

    return correct.mean(), correct.sum()


### Helper-function for showing the classification accuracy

def print_test_accuracy(show_example_errors=False,
                        show_confusion_matrix=False):

    # For all the images in the test-set,
    # calculate the predicted classes and whether they are correct.
    correct, cls_pred = predict_cls_test()

    # Classification accuracy and the number of correct classifications.
```

```
    acc, num_correct = classification_accuracy(correct)

    # Number of images being classified.
    num_images = len(correct)

    # Print the accuracy.
    msg = "Accuracy on Test-Set: {0:.1%} ({1} / {2})"
    print(msg.format(acc, num_correct, num_images))

    # Plot some examples of mis-classifications, if desired.
    if show_example_errors:
        print("Example errors:")
        plot_example_errors(cls_pred=cls_pred, correct=correct)

    # Plot the confusion matrix, if desired.
    if show_confusion_matrix:
        print("Confusion Matrix:")
        plot_confusion_matrix(cls_pred=cls_pred)


## Results


print_test_accuracy(show_example_errors=False,
                    show_confusion_matrix=False)


# In[44]:

optimize(num_iterations=1000)


# In[45]:

print_test_accuracy(show_example_errors=True,
                    show_confusion_matrix=True)


# In[46]:

optimize(num_iterations=1000)


# In[47]:

print_test_accuracy(show_example_errors=True,
                    show_confusion_matrix=True)
```

Tranfer Learning with Oxford Flower images

和文抄訳

今の時代では画像と動画といったマルチメディアの爆発的に増加している。フィルムカメラの時代はまだ人間が分類などをし管理することができていたが、携帯やコンパクトデジカメ、防犯カメラなどの普及により、コンピューターが私たちの代わりに人間が行う分類、管理を行う必要が出てきた。しかし、コンピューターが私たちと同様に多数の物体クラス分類することはまだしも、それをうまく説明するまでには至っていない。既存の物体クラス分類の研究にはニューラルネットワークが主に使われているが、そこで用いられている特徴量抽出はネットワークで自動生成したものか、簡易なものが多い。私たちは、そのような特徴量抽出では本当に必要な特徴は得られないという仮定を元に、類似画像捜索などで活用されている MPEG-7 の特徴抽出器を分析し、それを応用した色、模様、形などの特徴をニューラルネットワークと組み合わせることで、精度の高いネットワーク構築できるかを試みた。私たちは、ニューラルネットワークをグーグルが公開しているニューラルネットワークソフトウェアライブラリー”TensorFlow”を使い画像認識などでよく使われている”Convolutional Neural Network”や”Autoencoder”といったネットワークモデルを構築し、重みなどの変数を”Autoencoder”の事前学習の手法を使い最適化、また特徴量を抽出した画像をこの”Autoencoder”の入力画像とする実験を MNIST データセットなどで行った。そこから、輪郭などを抽出した画像の方が分類精度が上がったことから、そのあとのカラー画像の分類にはその外形を重点に置いた方向で実験を進めた。そこで、画像が大きく複雑な”Flower Datasets”に関しては輪郭ではなく物体の外形をうまく抽出できる”Ultrametritic Contour Maps”を活用し物体の外枠だけを抽出した。さらに、それを元画像の彩度を高めることにより新しい画像に変換した。その後、元の画像を学習対象とするネットワークと、新画像を学習対象とするネットワークの性能を比較し、外枠の部分の彩度

が高い新画像を使用した方が、わずかであるが元画像を使用した時よりも、精度の高い分類ができることが実験により示すことができた。