Sugar Cane Grading from photo using machine learning
2559:39

Miss Pham Thi Mai Phuong, Phuong, 56070503447, phuongmaipham@icloud.com

Advisor: Dr. Sally E. Goldin

September 29, 2016

_____

I have read this report and approve its content.

Abstract (English)

This project will work in co-operation with staff from Mitrphol Sugar Company to create a software test bed for improving sugarcane quality control over a large area. The test bed will be able to analyze the sugar cane health from mobile phone photos. The project will use machine learning techniques to train the software to discriminate photos based on cane quality. The test bed will first extract the sugar cane crucial features from mobile phone photos. This test bed will then classify the cane photos into different health categories based on extracted features using supervised machine learning. The results obtained from this project will be useful for developing a real world system to allow individual farmers to send photos of their fields, which can be analyzed and classified to get more detailed information about cane health over a wide area. This project is thus important because it will help sugar companies gain better information with a lower surveying cost.

Acknowledgements

This project cannot be completed without the support from …..

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Problem Statement and Approach

Sugarcane is an important crop in Thailand. In order to forecast sugarcane yeild, sugar companies need detailed information on the cane conditions in different fields. Features such as color and size of the leaves are some indicators of cane conditions (cane health).

A field's condition, such as soil properties, seed quality or irrigation system, has some important consequences for those features. However, not every field is identical. Thus, the features and the resulting can quality vary from one field to another. This will cause a problem in collecting data: there are too many fields and it is too complicated and time consuming for sugar companies to do exhaustive surveys to get the information that they need.

To address to this problem, this project will work in co-operation with staff from Mitrphol Sugar Company to create a software test bed for improving cane quality control over a large area. The test bed will be able to analyze the sugar cane health from photos of the fields taken from ground level. The project will use machine learning to train the software to discriminate photos based on cane quality. The results obtained from this project will be useful for developing a real world system to allow individual farmers to send photos of their fields to Mitrphol which can be analyzed and classified to get more detailed information about cane health over a wide area. This project is thus important because it will help sugar companies gain better information with a lower surveying cost.

This is a research - real world stakeholder project.

1.2 Objectives

The goal of this project is to develop a software test bed for experimenting with images of sugar cane using supervised machine learning technique. The test bed will first extract the sugar cane crucial features from photos. This test bed will then classify the cane photos into different health categories based on extracted features using supervised machine learning.

1.3 Scope

This project does not attempt to deliver a final system, but rather a software testbed. The testbed is intended to experiment with images of sugarcane with different machine learning models and conclude which model can achieve the best result.

Deliverables for Term 1

- ✓ Experimental data set
- ✓ Experimental design
- ✓ Some prototype using the selected framework
- ✓ Decision on what learning framework(s) to use, with justification

Deliverables for Term 2

- ✓ Complete experimental design of the test bed
- ✓ Software test bed with desirable results
- ✓ Results and data analysis

1.4 Tasks and Schedule

1.4.1 Task breakdown

1. Analyze and determine the requirements of the project
2. Plan the project schedule
3. Work on introduction chapter of the report (chapter 1)
4. Research emphasizing on the following topics:
   i. Work by other researchers on discriminating between similar images using machine learning
   ii. Machine learning methods for image classification and the available libraries
   iii. Basic image processing concepts
5. Create the project proposal and get feedback
6. Test prototypes for various learning frameworks and make a decision on which learning framework to use
7. Collect and create dataset
8. Study and understand the dataset
9. Create experimental design
10. Complete progress report for the first semester
11. Work on theory and background chapter of the report (chapter 2)
12. Work on methodology chapter of the report (chapter 3)
13. Prepare for presentation for the first semester
14. Write software to standardize and augment images
15. Write scripts to control the experiments
16. Test the system and fix bugs
17. Train and test the system with different parameters
18. Analyze the results
19. Complete final report for the second semester (Result + conclusions, chapter 4 and 5)
20. Create poster and prepare for presentation for the second semester

1.4.2 Draft Schedule

Step	Operation	Project Duration																																			
		2016																				2017															
		August				September				October				November				December				January				February				March				April			
		1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
1	Analyze and determine the requirements of the project																																				
2	Plan the project schedule																																				
3	Work on Introduction chapter of the report																																				
4	Research emphasizing on the following topics:																																				
	1) Work by other researchers on discriminating between similar images using machine learning																																				
	2) ML method for image classification especially CNN and the available libraries																																				
	3) Basic image processing concept																																				
5	Create the project proposal and get feedbacks																																				
6	Test prototypes for various learning frameworks and make a decision on which learning framework to use																																				
7	Collect and create dataset																																				
8	Study and understand the dataset																																				

9	Create experimental design																																								
10	Complete progress report for the first semester																																								
11	Prepare for presentation for the first semester																																								
12	Work on Background and Theory chapter of the report																																								
13	Work on Methodology chapter of the report																																								
14	Write pre-processing software to standardize images																			H	H	H																			
15	Write scripts to control the experiments																																								
16	Test the system and fix bugs																																								
17	Train and test the system with different parameters																																								
18	Analyze the results																																								
19	Complete final report for the second semester																																								
20	Create poster and prepare for presentation for the second semester																																								

** H:holiday

Chapter 2

Background, Theory and Related Research

2.1 Digital Image Processing Concepts

2.1.1 Digital Image Representation

Pixels are basic elements of an image. Each pixel depicts a light intensity value, which is represented as a number. Digital image is a collection of pixels. There are two types of digital image: grayscale image and color image. Grey scale image is defined by a matrix of pixels, whereas color image is usually defined by a cube made of three matrixes of pixels.

The density of pixels in an image is called 'resolution'. If we have an image of a particular scene, the level of detail will be controlled by the number of pixels. The more pixels that we keep to describe an image, the more detailed the image.

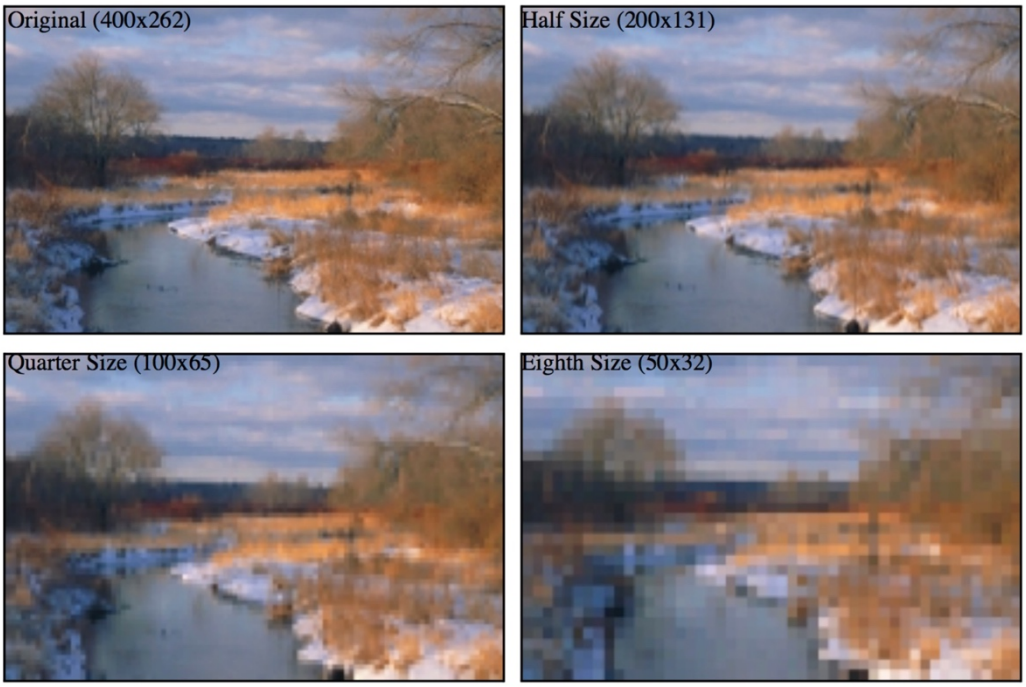

Figure 2.1-1 What happens as we reduce the resolution of an image while keeping its size the same. [9]

A monochrome image can be described in terms of a two-dimensional light intensity function f(x,y), where the amplitude of f(x,y) is the intensity (brightness) of the image at position (x,y). The intensity of a monochrome image lies in the range

$$L_{min} < f(x,y) < L_{max}$$

Equation 2.1-1

where the interval $[L_{min}, L_{max}]$ is called the grey scale. There are two common grey scale storage methods: 8-bit storage and 1-bit storage.

The most common storage method is 8-bit storage. It can represent up to 2^8colours for each pixel. The grey scale interval is [0,255], with 0 being black and 255 being white.

The less common storage method is 1-bit storage. There are two grey levels, with 0 being black and 1 being white. Image produced by this method is also called binary image.

Figure 2.1-2 Level grey scale. [9]

A colored image can be represented using multi-channel color models. The most widely used model is the RGB. Colored images are made up of three primary spectrums: red, green and blue (RGB). These three primary spectrums together create a three-dimensional color space with red defining one axis, green defining the second, and blue defining the third. Every existing color is described as a mixture of red, green, and blue light and located somewhere within the color space.

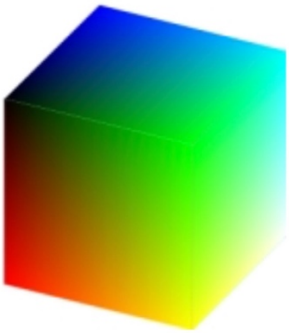

Figure 2.1-3 Color space. [9]

Using RGB model, a colored image can be described using a three-channel intensity function

$$I_{RGB}(x,y) = (F_R(x,y), F_G(x,y), F_B(x,y))$$

Equation 2.1-2

where $F_R(x,y)$ is the intensity at position (x,y) in the red channel, $F_G(x,y)$ is the intensity at position (x,y) in the green channel, and $F_B(x,y)$ is the intensity at position (x,y) in the blue channel. Each channel usually uses an 8-bit storage which can describe up to 2^8 colours. Thus, computers commonly use a 24-bit storage to describe the intensity at position (x,y), which can describe up to 2^{24} colours.

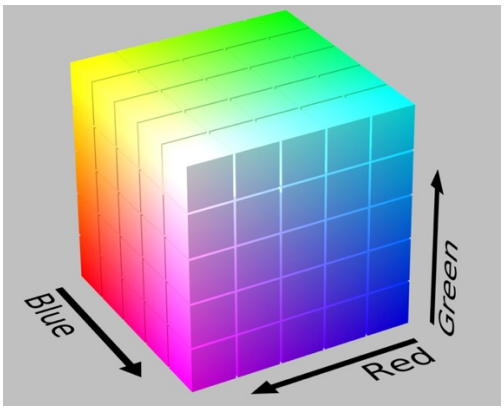

Figure 2.1-4 Color intensity. [11]

2.1.2 Image Processing Operations

Image operations are calculations applied to each pixel (x,y) of an input image f to transform the input image f(x,y) into an output image g(x,y). Different operation types are applied to an image depending on what kind of results we want to achieve. In this section, we will focus on discussing three types of image operations: resampling, cropping and histogram modifications.

Resampling is a technique used to generate a new image with a different resolution. Increasing the number of pixels is called upsampling and decreasing the number of pixels is called downsampling. Upsampling can sharpen the original image whereas downsampling softens or blurs the original image. There are several different resampling techniques. The main concept of those techniques is to interpolate the value of a new pixel from the existing pixels of the original image. Nearest neighbor resampling is the simplest method. The value of each pixel in the output image is the value of its nearest neighbor in the original image.

Cropping refers to the removal of the regions in the image. A new pair of (x,y) coordinates is required to define the corners of the cropped image. The resulting image is a smaller image that contains useful data for further studies.

Histogram is the frequency distribution, often portrayed as a graph but this is not required. Histograms are frequently visualized as graphs. An image's histogram measures the frequency of each pixel intensity in an image. For example, the histogram below shows the frequency of pixel values of a grey scale image, with which the x axis indicates the range of the pixel values and y axis indicates the frequency these values appear on the entire image.

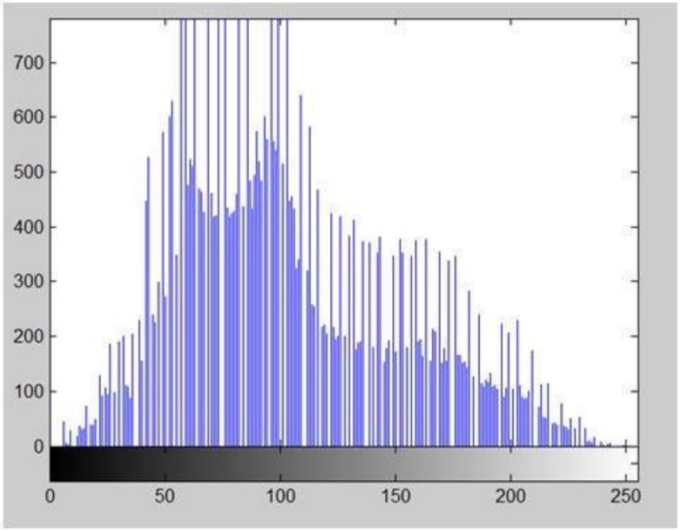

Figure 2.1-5 An image histogram[14]

There are various operations to modify an image's histogram. In this section we will focus on discussing about histogram sliding and histogram stretching.

Histogram sliding is a method used to change image brightness. To increase the brightness, we shift an image's histogram to the right by adding a constant value to every pixel of an image. In contrast, to decrease the brightness, we shift an image's histogram to the left by subtracting a constant value from every pixel of an image.

Histogram stretching is a method used to increase the contrast. Contrast is the different between the maximum and the minimum intensity values of an image. There are several ways to do histogram stretching. We will illustrate how histogram stretching works by demonstrating one of them through the following formula:

$$g(x,y) = \frac{f(x,y) - f_{min}}{f_{max} - f_{min}} \, 2^{bpp}$$

Equation 2.1-3

where $f(x,y)$ is the pixel intensity of the original image, f_{min} is the minimum pixel intensity, f_{max} is the maximum pixel intensity and bpp is the bit per pixel. The purpose of this formula is to 'spread' the image's old intensity values over a larger range. By doing so, we increase the difference between the maximum and the minimum intensity values of an image and hence increase the contrast.

2.2 Image Convolution

2.2.1 Concepts and Mathematics

Convolution is an image transformation technique using neighborhood (or area-based) operators. The objective of image convolution is to process an input image in order to enhance some important features that are suitable for a specific application. Convolution has two main approaches: spatial-domain approach and frequency-domain approach. Spatial-domain and frequency-domain approach can both be described in terms of convolutional function:

$$g(x,y) = h(x,y)*f(x,y)$$

Equation 2.2-1

given f(x,y) is the input image, h(x,y) is a small pattern called the kernel and g(x,y) is the output image.

In spatial-domain, we look at the value of each pixel varies with respect to scene whereas in frequency-domain, we look at the rate at which the pixel values change in spatial-domain. Thus, in frequency-domain approach, we will have to convert the pixel values into frequency-domain before applying convolution, then convert the result back into spatial-domain. We will mainly discuss about spatial-domain procedure in this section.

The procedure for convolution in spatial-domain is as follows: A filter (sometimes can be referred to as a mask, a kernel or a window) is slid around the input image. The filter is a smaller matrix of values called coefficients. Each time the filter is placed at a new position, every pixel of the input image contained within the filter is multiplied by the corresponding filter coefficient and then summed together. The result from each multiplication and summation becomes the value of the next pixel of the output image. The described procedure is repeated this until all values of the image has been calculated.

2.2.2 Examples – Sobel, Blurring

Below we will give two simple examples to illustrate the application of convolutional filtering.

The first application is to highlight edges in an image. Some well-known edge enhancement filters are Prewitt operator, Sobel operator, Robinson compass masks, Krisch compass Masks and Laplacian operator. The decision on which filter to use depends on our desired results. Here we will describe the use of Sobel edge detection. A Sobel filter is used to calculate edges in both horizontal and vertical direction.

The vertical mask of Sobel operator is as follows:

-1	0	1
-2	0	2
-1	0	1

Table 2.2-1 Vertical Mask of Sobel operator

The pixels at the corresponding position to the area declared by this filter of the input image are respectively multiply by -1,0,1,-2,0,2,-1,0 and 1. The results of these nine multiplications are then summed. The filter is then moved to the next position in the image. This process is repeated until all values of the image has been calculated.

The horizontal Mask of Sobel operator is as follows:

-1	-2	-1
0	0	0
1	2	1

Table 2.2-2 Horizontal Mask of Sobel operator

Similarly, the pixels at the corresponding position to the area declared by this filter of the input image are respectively multiply by -1,-2,-1,0,0,0,1,2 and 1. The results of these nine multiplications are then summed. The filter is then moved to the next position in the image. The horizontal and vertical output images are kept separately. This process is repeated until all values of the image has been calculated.

This filter gives more weight to the pixel values around the edge region. It thus increases the edge intensities. As a result, the output image edges become enhanced comparatively to the original image.

The second application is to blur an image. There are three common type of filters that are used to perform blurring: mean filter, weighted average filter and Gaussian filter. We are going to discuss about mean filter. In a mean filter, there are an odd number of filter elements, all of which are the same and which sum to one. For example, a 3x3 mean filter can be declared as followed:

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

Table 2.2-3 3x3 Mean filter

The pixels at the corresponding position of the input image to the area declared by this filter are multiplied by 1/9. The results of these nine multiplications are then summed. This process is repeated until all values of the image has been calculated.

This operation can be used to suppress noise data that may be present in a digital image as a result of poor sampling system or transmitting channel such as isolated bright pixels.

2.3 Machine learning

2.3.1 Core concepts of machine learning

Machine learning studies computers' abilities to automatically recognize patterns and generate intelligent conclusions from the given data. It is generally divided into two learning types: supervised and unsupervised.

Supervised learning (SVM) is equivalent to data classification. Computers learn patterns from the labeled examples then use them to make intelligent classification on the unknown data.

Unsupervised learning (USVM) is equivalent to clustering. Initially, there is no label associated with the data. Computers try to divide the dataset into clusters to discover classes based on pre-existing structure within the data. USVM cannot narrate semantic meaning of the clusters in their learnt models.

Data classification consists of a learning step and a classification step. In the learning phrase, computers build a classification model by studying the **training set** made up of pre-labeled tuples. A tuple X describes a set of features. Each tuple is represented by a vector of n-dimension X = (x_1, x_2,..., x_n) where x_i is a value for one feature dimension, and categorized with a class label attribute x_c. The class label attribute has a discrete value that indicates which class tuple X tuple belongs.

In the classification phrase, the learned model is used for predicting the class label for new data. To evaluate this model, a **testing set** is used. Each testing set is also made up of pre-labeled tuples. Testing data sets are usually independent of training sets.

When the resulting model is tailored to fit one specific sample rather than reflecting the overall population, we have an **overfitting** problem. There are various methods to reduce **overfitting**. One is to put some random noise in the data. Another is to stop training before 100% accuracy is achieved.

2.3.2 Training and testing strategies

The given dataset is divided into training and testing sets. There are various strategies for doing this, including the three following methods:

Hold out: The given data set is divided into two independent sets: training set and testing set. Usually two-thirds of the data are in the training set and one-third of the data is in the testing set.

k-fold cross-validation: The data set is divided into k subsets, and the holdout method is repeated k times. Each time, one of the k subsets is used as the test set and the other k-1 subsets are put together to form a training set. Then the average error across all k trials is computed. The advantage of this method is that it matters less how the data gets divided.

Bootstrap: A common bootstrap method is the .632 bootstrap. Given a dataset of d tuples, this dataset will be sampled d times with replacement. Each time a tuple is selected, it is back-added into the data pool and may be selected again. The data that do not make it into the training set will eventually be added into the testing set. The training set and the testing set are not independent because the same tuple can be included in both the training and the testing sets. In .632 bootstrap, 63.2% of the dataset will end up in the bootstrap sample, and the 36.8% that did not make it to the bootstrap sample will together form the test set. Bootstrapping is especially useful when we want to replicate the distributional properties of a random variable in the real world since in the real world, it is not always the case that a tuple only appears once.

2.3.3 Model evaluation

Confusion matrix is often used to measure the performance of our classifier. A confusion matrix is represented by a table of size jxj, where j is the number of output classes. A table entry a_{ij} is in row i^{th} and column j^{th}, indicates the number of tuples that are actually in class i but were classified by the classifier to be in class j.

For calculating convention, for every output class j, we will categorize the the tuples classified by the classifier into four groups, as suggested by Jiawei et al. [8]

True positive (TP): For each given class j, tuples that are pre-labeled as class j and correctly recognized by the classifier to be in class j are true positive. TP represents the number of true positive tuples.

True negative (TN): For each given class j, tuples that are pre-labeled as class i and correctly recognized by the classifier to be in class i are true negative. TN represents the number of true negative tuples.

False positive (FP): For each given class j, tuples that are pre-labeled as class i and mistakenly recognized by the classifier to be in class j are false positive. FP represents the number of false positive tuples.

False negative (FN): For each given class j, tuples that are pre-labeled as class j and mistakenly recognized by the classifier to be in any class other than j are false negative. FN represents the number of false negative tuples.

Table 2.3-1 and 2.3-2 shows two examples of TP, TN, FP, FN for class 0 and class 1 respectively.

	Predicted class				
Actual class	0	1	2	…	j
0	TP	FN	FN	FN	FN

1	FP	TN	FN	FN	FN
2	FP	FN	TN	FN	FN
…	FP	FN	FN	TN	FN
j	FP	FN	FN	FN	TN

Table 2.3-1 Confusion matrix for class 0

	Predicted class				
Actual class	0	1	2	…	j
0	TN	FP	FN	FN	FN
1	FN	**TP**	FN	FN	FN
2	FN	FP	TN	FN	FN
…	FN	FP	FN	TN	FN
j	FN	FP	FN	FN	TN

Table 2.3-2 Confusion matrix for class 1

By dividing classified tuples into these four groups, we can now evaluate our classifiers. The classifier accuracy or recognition rate on the given test set is calculated as follow:

$$accuracy = \frac{TP + TN}{P + N}$$

Equation 2.3-1

where

$$P = TP+FP$$

Equation 2.3-2

and

$$N = TN+FN$$

Equation 2.3-3

Similarly, the classifier error rate or misclassification rate is:

$$Error\ rate\ = 1 - accuracy = \frac{FP + FN}{P + N}$$

Equation 2.3-4

2.4 Multilayer feed-forward neural networks (MFNN)

2.4.1 Overviews

MFNN is a classic machine learning algorithm that simulates the way human brain

works. MFNN is very useful for studying large datasets due to its ability to tolerate noise data as well as to learn patterns without prior knowledge. It is especially efficient for real world data, where we do not know much about the relationships between attributes and classes. Generally, MFNN requires a large amount of data in order to well perform.

Each MFNN consists of an input layer, one more hidden layers and an output layer. Layers are connected in acyclic graph. Each layer is made up of computational elements called **neurons**. Neurons between two adjacent layers are pairwise connected, but neurons within one layer share no connection. No direct connection exists between input and output layer.

Inputs are fed into the neurons making up the input layer. The outputs produced by this layer are weighted and passed simultaneously to the first hidden layer. This hidden layer outputs are again weighted and sent to an another hidden layer and so on. It is arbitrary how many hidden layers there should be. The weighted outputs of the last hidden layer are sent to the output layer, where the prediction for the given tuples will be produced.

Neurons in the input layer are **input units**. Neurons in the hidden layers and the output layers are called **neurodes** or sometimes referred to as **output unit**s. The number of output units are not necessarily equal number of input units. There can be more or less number of hidden units than number of input or output units. Each output unit applies a nonlinear (activation) function to its input. The *activation function* will be described in section 2.5.6.

In a traditional MFNN, all neurons in layer j are connected to each neuron in layer k, for all layers. Thus we say that the NN is fully connected, as shown in figure 2.4-1.

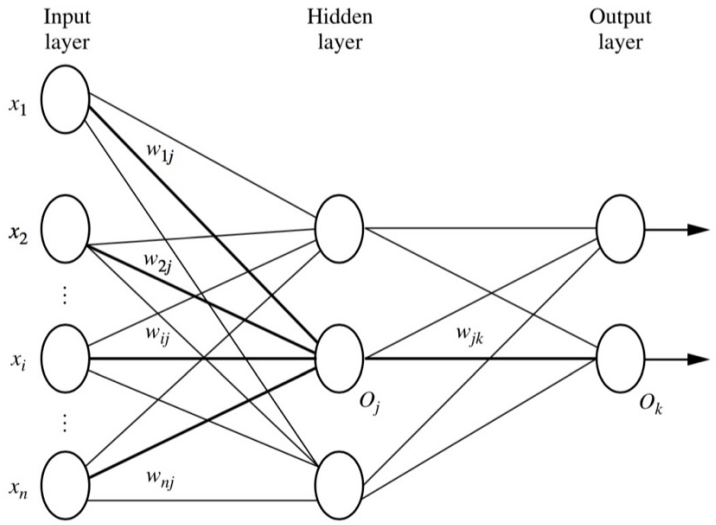

Figure 2.4-1 Fully connected layers. [8]

2.4.2 Back propagation

Back propagation is the learning algorithm of neural networks. It propagates the total error at the output layer in the backward direction. During that procedure, it updates the weights on each connection based on calculations using this error so that in the end the error will be reduced.

The true output of any neuron k is calculated as follows. Each connection from neuron j at layer (l-1) to neuron k at layer l is associated with a weight. The output from neuron j is multiplied by the weight between the connection between j and k, summed across all connections entering neuron k. The result is summed with the bias of neuron k layer l.

Weight raises the effect of the connection to a neuron whereas bias raises the importance of all connections to one neuron:

$$a_k^l = f(\sum_{j=1}^{m} w_{jk} a_j^{l-1} + b_k^l)$$

Equation 2.4-1

Where f is the neural activation function, a_k^l is true output of neuron k at layer l, a_j^{l-1} is true output of neuron j at layer (l-1), w_{jk} is the weight of the connection between node j and k, b_k^l is the bias of node k.

Total error is the summation of error at each output neuron. The error at each neuron is a function of difference between the actual value of that neuron pre-labeled in the training set and the value predicted by the neural networks. It can be represented in the following equation:

$$E = f(y_{o_N}^o - a_{o_N}^o)$$

Equation 2.4-2 [19]

where $y_{o_N}^o$ is the actual label of a tuple, $a_{o_N}^o$ is the label of that tuple predicted by the neural networks and f is any *loss function*. According to many papers and articles [1,2,3,4,5,6,19,20], the loss function used to calculate total error in neural networks is mean squared error, whereas in convolutional neural networks it is cross-entropy error.

We want to update the current weight w_{ij}^l at each connection between neuron i and neuron j in the networks by an amount of Δw_{ij}^l in order to reduce this error:

$$w_{ij}^l(\text{new}) = w_{ij}^l(\text{old}) \pm \eta \Delta w_{ij}^l$$

Equation 2.4-3 [19]

where η is the *learning rate*, w_{ij}^l (old) is the current weight between neuron i and neuron j at layer l, w_{ij}^l (new) is the updated weight between neuron i and neuron j at layer l and Δw_{ij}^l is the amount of weight to be updated at layer l in order to reduce the total error. The larger the learning rate, the faster the networks will learn. Typically η is set to 0.5 to avoid premature convergence.

The amount of weight to be updated at each connection is the rate at which the total error changes with respect to the change of weight of each connection. This rate can be calculated by taking the derivative of the total error with respect to weight. However, we have not yet known the output of this derivative, therefore, we need to take a step further by applying the chain rule in order to calculate the derivative value from something that we can possibly know:

$$\Delta w_{ij}^l = \frac{\partial E}{\partial w_{ij}^l} = \frac{\partial E}{\partial a_j^l} * \frac{\partial a_j^l}{\partial z_j^l} * \frac{\partial z_j^l}{\partial w_{ij}^l}$$

Equation 2.4-4 [19]

where Δw_{ij}^l is the amount of weight to adjust between neuron j and neuron i at layer l, a_j^l is the *true output* of neuron j at layer l, z_j^l is the *net output* of neuron j at layer l, w_{ij}^l is the current weight between neuron j and neuron i at layer l. The net output of a neuron is the output before applying activation function. The real output a neuron is the output after applying activation function.

We can now calculate the value of Δw_{ij}^{l} by calculating the right hand side of equation 2.4-4 separately in two parts, $\frac{\partial z_j^l}{\partial w_{ij}^l}$ and $\frac{\partial E}{\partial a_j^l} * \frac{\partial a_j^l}{\partial z_j^l}$ as follows:

The value of $\frac{\partial z_j^l}{\partial w_{ij}^l}$ can be calculated by taking the derivative of the net output of neuron j at layer l with respect to the the current weight between neuron j and neuron i at layer l. Since the net output of neuron j at layer l is

$$z_j^l = \sum_{i=1}^{m} w_{ij} a_i^{l-1} + b_j^l$$

Equation 2.4-5

which is equation 2.4-1 before applying activation function, the result from this derivative is a_i^{l-1}, which is the sum of true output of all neuron i at layer (l-1) that are connected with neuron j at layer l.

We will denote $\frac{\partial E}{\partial a_j^l} * \frac{\partial a_j^l}{\partial z_j^l}$ as $\delta_{z_j}^l$. It is in fact $\frac{\partial E}{\partial z_j^l}$ which is the rate at which the total error changes with respect to the change of net output of neuron j at layer l. The value of $\delta_{z_j}^l$ cannot be calculated directly. Further chain rule needs to be applied to this term:

$$\delta_{z_j}^l = \frac{\partial E}{\partial a_j^l} * \frac{\partial a_j^l}{\partial z_j^l} = \frac{\partial E}{\partial z_i^{l+1}} * \frac{\partial z_i^{l+1}}{\partial a_j^l} * \frac{\partial a_j^l}{\partial z_j^l}$$

Equation 2.4-6 [19]

where a_j^l is the true output of neuron j at layer l, z_j^l is the net output of neuron j at layer l and z_i^{l+1} is the net output of neuron j at layer (l+1).

To calculate the terms $\frac{\partial a_j^l}{\partial z_j^l}$, we will need to understand about the activation function. This will be explained in section 2.5.6. The activation function of each neuron j is a function of net output at neuron j. Therefore, the derivative of true output with respect to net output can be calculated directly.

The term $\frac{\partial E}{\partial z_i^{l+1}} * \frac{\partial z_i^{l+1}}{\partial a_j^l}$ is in fact the summarization of the multiplication between the rate at which the total error changes with respect to the change of net output of each neuron i at layer (l+1) that are connected to neuron j at layer l and the weight between neuron j and neuron i.

Therefore, $\delta_{z_j}^l$ can be written as:

$$\delta_{z_j}^l = \left(\sum_{0 \le i \le m}^{l} \delta_{z_i}^{l+1} * w_{ji}^{l+1} \right) * \frac{\partial a_j^l}{\partial z_j^l}$$

Equation 2.4-7

where $\delta_{z_j}^l$ is the rate at which the total error changes with respect to the change of net output of neuron j at layer l, $\delta_{z_i}^{l+1}$ is the rate at which the total error changes with respect to the change of net output of neuron i at layer (l+1), w_{ji}^{l+1} is the weight between neuron j and neuron i at layer (l+1), a_j^l is the true output of neuron j at layer l and z_j^l is the net output of neuron j at layer l.

The rate at the outer most layer (the output layer) can be calculated directly because the total error is a function of true output of all neurons at output layer.

The above equations show that the networks can back-propagate the error rate from the outer most layer (output layer) to the inner most layer (input layer). At the same time, it uses that error rate to update weight at each connection in the networks in order to reduce the total error.

2.5 Convolutional neural networks (CNN)

2.5.1 Overviews

CNN is a "state-of-the-art technique for image recognition" [5]. It is a multilayer neural networks (as illustrated in figure 2.5-1). It consists of one or more convolutional layers, followed by pooling layers (sometimes called subsampling layers), ReLU layers, and one or more fully connected layers. Convolutional layers are used for extracting important features from the input images. The features learnt by a convolutional layer are often summarized by a pooling layer. ReLU layer eliminates negative outputs produced by the pooling layer preceding it. The learnt features are eventually passed into fully connected layers, where each input image is mapped with a suitable output class.

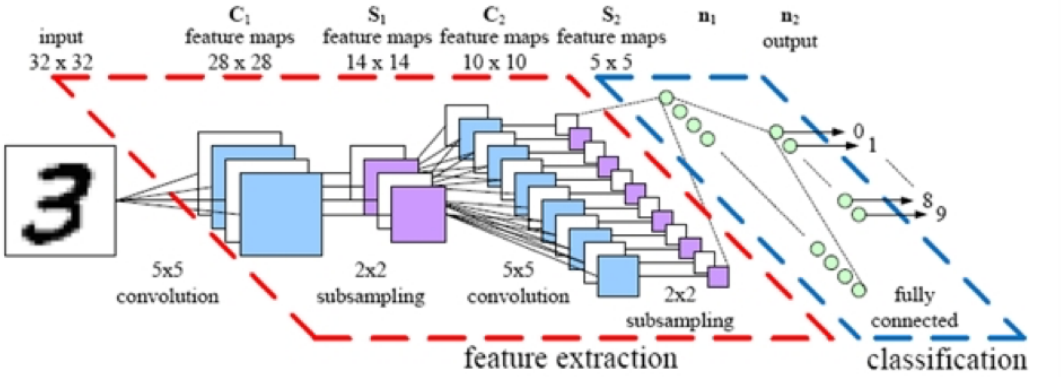

Figure 2.5-1 A convolutional neural networks. [16]

The networks build a sufficient model for the input dataset by imposing a set of forward and back propagations. At each iteration, the CNN model tries to reduce the number of wrong predictions by updating the weights that are associated with the location of the errors. The updating process iterates until all the weights in the network converge.

2.5.2 Convolutional layer

Convolutional layer extracts the important features of images via image convolution. Four parameters are required in convolutional layer: the number of filters K, receptive field size F, the stride S and the amount of zero padding P. This layer accepts input of volume size $[Width_{in}][Height_{in}][Dimension_{in}]$ and produces an output of volume size $[(Width_{in} - F + 2P)/S + 1][(Height_{in} - F + 2P)/S + 1][K]$.

2.5.2.1 Local receptive field

Think of the input image as a square of n x n neurons. The value of each neuron is the corresponding pixel intensity of the input image. We will map a localized region of the input neurons to a neuron in the hidden layer. These localized regions of input neuron are called local receptive fields. The size of a local receptive field is equal to the size of a kernel. More about kernel will be discussed in section 2.5.2.2.

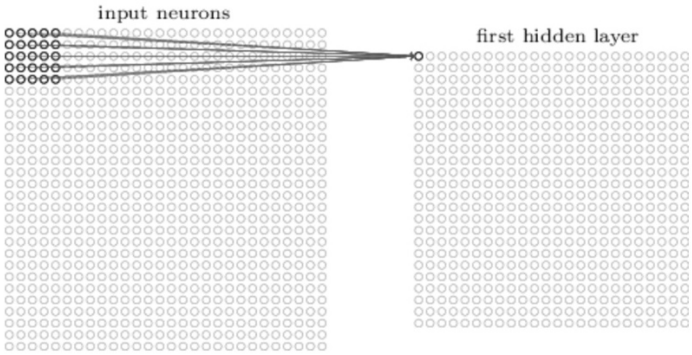

Figure 2.5-2 Local receptive fields [7]

2.5.2.2 Filter (Kernel)

Each mapping from input layer to a hidden neuron at the hidden layer next to it learns a weight $w_{a,b}^{l}$ and each corresponding hidden neuron learns a bias $b_{x,y}^{l}$. This weight and bias together make up a kernel (sometimes called a filter). In an FxF filter, each of the F positions holds weight which is a real value. The bias applies to the filter as a whole. As discussed in section 2.7, the filter is moved around the input image and the output pixel is calculated using the convolution operation. The output at position x^{th}, y^{th} is a convolutional function between an input neuron at position $(x-a, y-b)^{th}$'s value and its weight:

$$a_{x,y}^{l} = \sigma\left(\sum_{k}\sum_{k} w_{a,b}^{l}\, a_{x-a,y-b}^{l-1} + b_{x,y}^{l}\right)$$

Equation 2.5-1

where σ is neural *activation function*, $a_{x,y}^{l}$is the true output value at position (x,y) at layer l, $a_{x-a,y-b}^{l-1}$ is the input pixel value at position (x-a, y-b) at layer (l-1), (a,b) is position of a pixel in the kernel, $b_{x,y}^{l}$ is the the bias, $w_{a,b}^{l}$ the weight of the connection at the a^{th} row and the b^{th} column of the kernel and k is the size of the kernel. More about the activation function will be described in section 2.5.6.

2.5.2.3 Feature map and stride

The combination of outputs generated by a hidden layer is called a feature map. To calculate the value of the next pixel in the feature map, we will move our kernel k pixels sequentially along width and height. In this case we say a stride length of k is being used. Then we re-apply the convolution function discussed above, until we run out of input neuron.

Figure 2.5-3 shows an example of the convolutional process using two 3×3x3 convolutional kernels with stride 2. The first pixel of the first output volume is calculated as follows. The first channel of filter W_0 is convolved with the first 3x3 receptive field of the first input channel. The second channel of filter W_0 is convolved with the second 3x3 receptive field of the second input channel. The third channel of filter W_0 is convolved with the third 3x3 receptive field of the third input channel. Corresponding pixel values in the results from these three convolutions are then summed together. Then the bias b_0 of filter W_0 is added to each output pixel value. The next pixels of first output volume are calculated similarly with one variation: the aforementioned receptive field is slid to the right or down by 2 pixels each time.

Likewise, the pixels of second output volume are calculated with the same fashion but the filter W_0 is replaced by filter W_1.

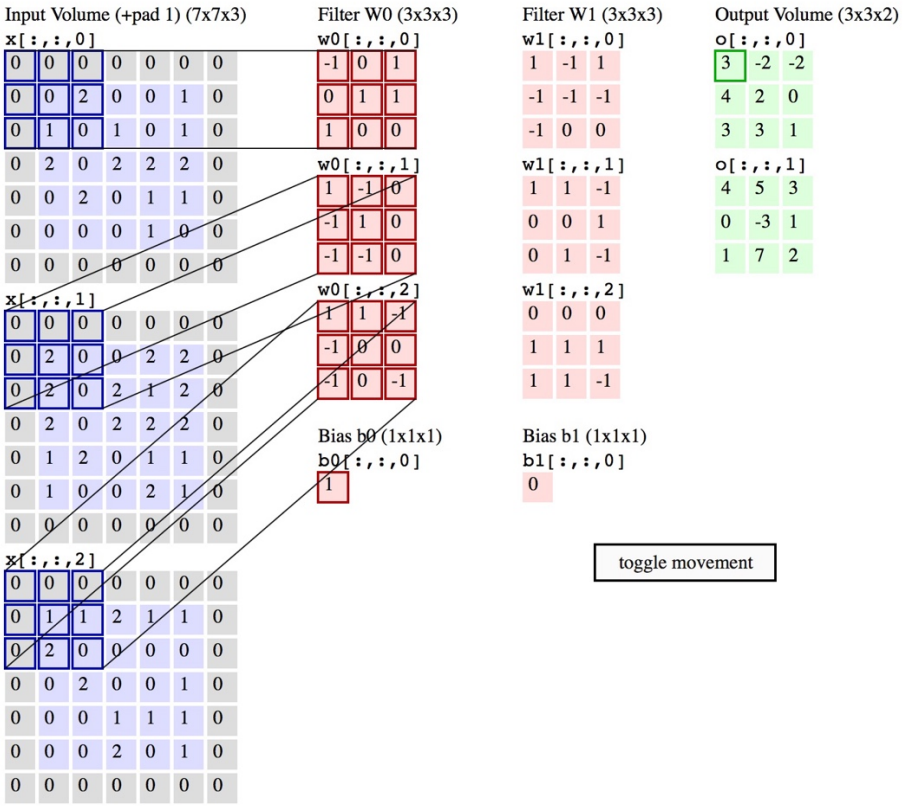

Figure 2.5-3 Convolutional process - An input image of size 7x7x3 is filtered by 2 convolutional kernels which create 2 feature maps. [12]

Each feature map can detect one feature at different locations of the same input image. To detect many features from one input image, we need several feature maps. To generate several feature maps, we will need several kernels. A complete convolutional layer consists of several different feature maps.

2.5.2.4 The use of zero-padding

Zero padding is a simple method of padding the borders of the input image in order to control the size of the output image. Padding allows convolution to be performed on pixels at the edges of the input, which otherwise would not have complete receptive fields. Specifically, the relationship among padding size P, input height/length W, output height/length O and filter size K can be described as follow:

$$O = \frac{W - K + 2P}{2} + 1$$

Equation 2.5-2

2.5.3 ReLU non-linearity

Rectified Linear Unit is a modern way to represent an output f as a function of input x. It computes the function:

$$f(x) = \max(0,x)$$

Equation 2.5-3

where

$$x = w*a + b$$

Equation 2.5-4

which means the output is 0 when the input is less than 0 and the output is linear with slope 1 when x the input is greater than 0.

ReLU has two advantages over *tanh* function which was traditionally used in MFNN as an output function. Deep learning neural networks with ReLU was found to train significantly faster than networks with tanh unit. This is because saturating nonlinearities are generally slower than non-saturating nonlinearities in terms of training time with gradient descent. Secondly, while tanh function involves expensive operations, ReLU can be implemented easily by thresholding a matrix of activations at zero. A convolutional layer is often followed by ReLU.

2.5.4 Pooling layer

A convolutional layer is usually followed by a pooling layer. The function of the pooling layer is to simplify every feature map in the previous layer at every depth slice. Thus it reduces the amount of parameters and computation in the network. As a result, it will reduce overfitting. Recall that overfitting happens when the model is tailored to fit the random noise in one specific sample. By reducing the amount and improving the quality of the data presented to the networks, overfitting can be reduced. A successful pooling layer should be able to preserve critical information while being "invariant to troublesome deformations" [9].

Pooling layer requires two parameters: filter size F and stride S. It accepts input with volume size $[Width_{in}][Height_{in}][Dimension_{in}]$ and produces an output of volume size $[(Width_{in} - F)/S + 1][(Height_{in} - F)/S + 1][Dimension_{in}]$.

There are several widely used pooling techniques. The most well-known one is **max pooling**. In max pooling, we want to know the max value of the pixels in a particular receptive field. This can be described as:

$$a^{l}_{x,y} = max_{0 \le p,q \le k}\{a^{l-1}_{x+p,y+q}\}$$

Equation 2.5-5

where $a^{l}_{x,y}$ is the output value of pooling layer l at position (x,y), $a^{l-1}_{x+i,y+j}$ is the true output value of convolutional layer (l-1) at position $(x + i, y + j)$ with (i,j) is position of a pixel in the filter.

Figure 2.5-4 shows an example of max pooling operation on four 4 x 4 slices. Each slice is filtered by a 2x2 kernel with stride 2 which results in four pooling feature maps, each of size 2 x 2. The first pixel of an output volume is the maximum value of the pixels in the first 2x2 receptive field of the corresponding input volume. The next pixels of that output volume are calculated similarly with one variation: the aforementioned receptive field is slid to the right or to the bottom by 2 pixels each time.

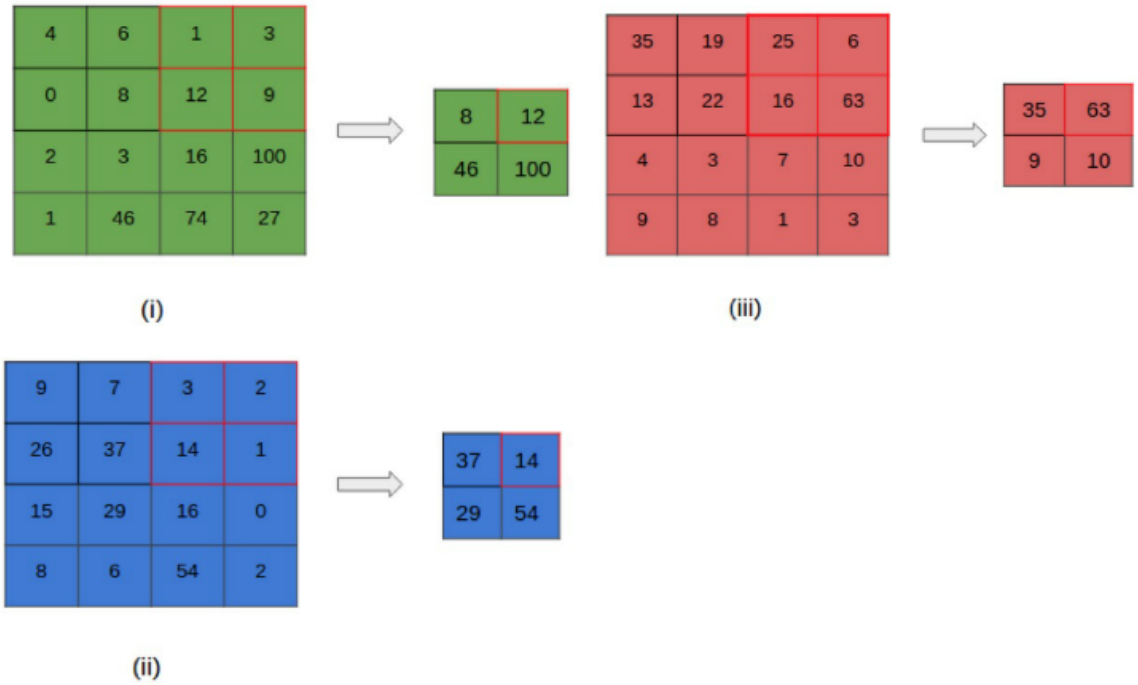

Figure 2.5-4 A max pooling layer [12]

Other popular pooling techniques are l_2 pooling and subsampling. Details about l_2 pooling can be found in Bruna et al. paper [21]. Details about subsampling can be found in Dominik et al. paper [18].

A filter size of 2x2 and a stride of length 2 are often applied with these pooling techniques. According to some research [4,9], there is no best pooling technique. We may need to try applying several pooling techniques to our problem to see which one yields the best result.

2.5.5 Fully connected layer

The output from the last pooling layer is flatten into 1-D vectors before it is sent to a set of fully connected layers. Fully connected layers are originated from multi-layers feed forward neural networks. Details on MFNN have been presented in section 2.4.

2.5.6 Softmax regression function

Softmax regression is a generalized logistic regression model which is used to turn a single neuron into a linear classifier so that the neuron can handle multiple classes. The CNN maps each tuple a_i^0 in the input layer to all neurons in the output layer and assign each output neuron to a probability a_i^l. The probability a_i^l is the true output of each neuron k at the output layer given the input tuple a_i^0, where each output neuron represents a class k. a_i^l gives the probability that tuple a_i^0 belongs to a class k,

The softmax regression function maps an input to an output in a range between 0 and 1, and all calculated probabilities of different classes must sum to 1. The true output a_i^l for each neuron k in the output layer can be computed by applying softmax regression function to the net output of every neuron k. Recall that the net output of a neuron is the output before applying softmax function and the real output a neuron is the output after applying softmax function.

For each output neuron, we perform the following calculation:

$$a_i^l = P\left(y_i^l = j \middle| a_i^0\right) = \frac{e^{(\sum_{i=1}^{m} w_{ij} a_i^{l-1} + b_j^l)}}{\sum_{k=1}^{K} e^{(\sum_{i=1}^{m} w_{ik} a_i^{l-1} + b_k^l)}} = \frac{e^{z_j^l}}{\sum_{k=1}^{K} e^{z_k^l}}$$

Equation 2.5-6

where y_i^l represents an output neuron j where j ϵ K, z_j^l is net output of neuron j, z_k^l is net output of each neuron k, where k = 1,…,K and K is the number of classes in output layer.

2.6 Learning in a Convolutional neural networks

2.6.1 Back propagation core concepts

Back propagation is an iterative process to adjust the learnt weight and bias by comparing the network's prediction with the tuple's known target value. It takes place after each training example is presented to the network. The aim is to minimize the error between the network's prediction and the known target value. The target value can either be a known class label or a continuous value. The weight and bias modification process is done in a backward direction, from softmax through fully connected, pooling and ReLU layers to the convolutional layer. The forward and backward process is repeated until a predefined termination condition occurs. The steps are as follows:

1. Initialize all weights and bias in the network with some small random values and choose a learning rate.
2. Propagate the inputs forward the networks using our initialized weights and bias until we reach the output layer.
3. Calculate error rate and consequently update the weight at each layer.
4. Repeat the process until when one of the following conditions is reached:
    - All Δw in the previous epoch are below some pre-specified threshold
    - A pre-specified number of epochs has reached
    - The percentage of misclassified tuples in the previous run is lower than some predefined threshold

2.6.2 Back propagation at the Softmax

Details about how to E_{total} calculate are presented in Section 2.4.2.

We will look at how the networks adjust its weight using E_{total} by looking at E_{total} effect on each layer.

2.6.3 Back propagation in the Fully connected layer (Neural networks)

Details about this can be found in Section 2.4.2.

2.6.4 Back propagation in the ReLU layer

The output a_j^l for each neuron can be written as

$$a_j^l = max(0, a_i^{l-1})$$

Equation 2.6-1

where a_i^{l-1} is the real output value of neuron i of ReLU layer (l-1).

Equation 2.6-11 shows that there is no weight associated with this layer. Thus this layer has no effect on the total error and the error rate at this layer is equal to the error rate at the fully connected layer.

2.6.5 Back propagation in the pooling layer

The output a_j^l for each neuron can be represented by

$$a_{x,y}^{l} = max_{0 \le p,q \le k}(a_{x+p,y+q}^{l-1})$$

as suggested by equation 2.5-5. Similar to ReLU layer, the above equation shows that there is no weight associated with the pooling layer. Therefore, pooling layer has no effect on the total error. The error rate at this layer is also equal to the error rate at fully connected layer.

2.6.6 Back propagation in the convolutional layer

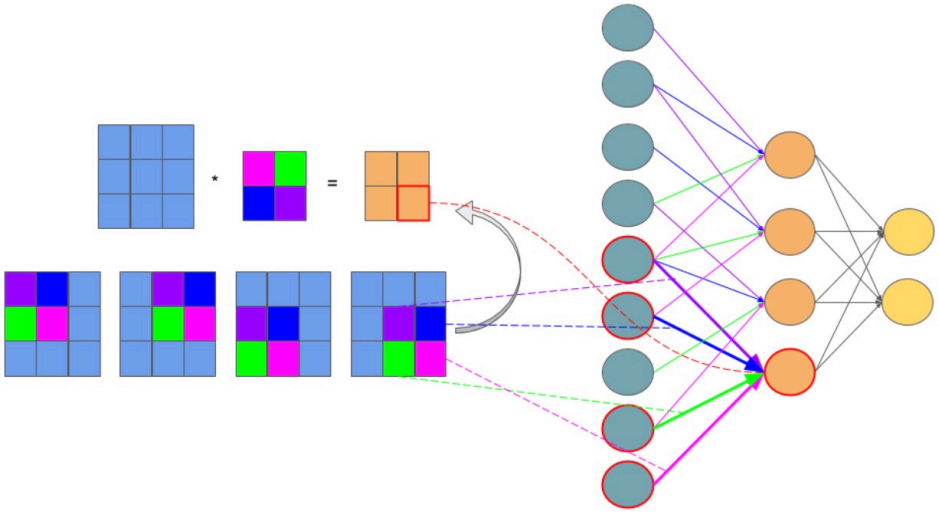

Figure 2.6-1 Feedforward in CNN is identical with convolution operation. [15]

Figure 2.6-1 illustrates that a feedforward in CNN is equivalent to the convolution operation. If we can somehow 'open' every component of a CNN in a 1-D vector, each pixel in the input image of a CNN will become a neuron in the input layer of a MFNN. Each pixel in a kernel of the CNN will become a weight of a connection between a neuron in the input layer and a neuron in the layer ahead of it, which can be a hidden layer or an output layer. Each pixel in the output image produced by convoluting the input image and a kernel in CNN will become a neuron in the hidden/output layer of a MFNN. And the bias of a kernel will become the bias of every neuron in the same hidden/output layer. The problem has now become similar to the case of MFNN. The error rate from the outer most layer (output layer) is back-propagated to the inner most layer (input layer). At the same time, the network uses that error rate to update weights at each connection in the networks in order to reduce the total error.

2.6.7 What does a CNN learn?

This section briefly discusses about the final CNN and how it represents learning. A CNN represents its learning as adapted filters at convolutional layers and weights at fully connected layers. Each filter at convolutional layer can detect a feature at different locations of the same input image. By learning a variety of selective filters, the networks improve its ability to detect robust features. Vice versa, examining learnt kernels can help us understand what kind of features are being extracted. Examining the kernels also helps us debug the problems with our model (for instance: Are the kernels informative enough?) and improve our results accordingly. Learning at fully connected layers on the other hand is represented with adapted weights at each connection. Those weights are the networks' reference when mapping features representing each object with the output class.

2.7 Data Augmentation

2.7.1 Concept

Artificially enlarging the dataset with label-preserving transformations is the simplest and the most known strategy to reduce overfitting on image data. To each randomly chosen sample

image, we will apply n *random transformations*. Each of these random transformations is a combination of several elementary forms of transformation, which we will describe shortly. The benefit of using transformations that require little computation is that of we will not have to store the pre-processed image on the disk. Image transformation contains two major approaches: point operators (sometimes called 1-to-1 pixel transforms) and neighborhood (or area-based) operators. In point operators, each output pixel value is strictly a function of the corresponding input pixel value. Brightness and contrast adjustments are two examples of such transformation. Techniques like convolution, which we have discussed in part 2.2, are not 1-to-1 transform.

2.7.2 Methods

2.7.2.1 Brightness adjustment

We add or subtract a constant amount of light to all input pixel in order to change an image brightness, as suggested in the following function:

$$g(i,j)= f(i,j)+\beta$$

Equation 2.7-1

where f(i,j) is the pixel located in the i-th row and j-th column of the input image, g(i,j) is the pixel located in the i-th row and j-th column of the output image and β is a bias parameter which is used to control the image brightness.

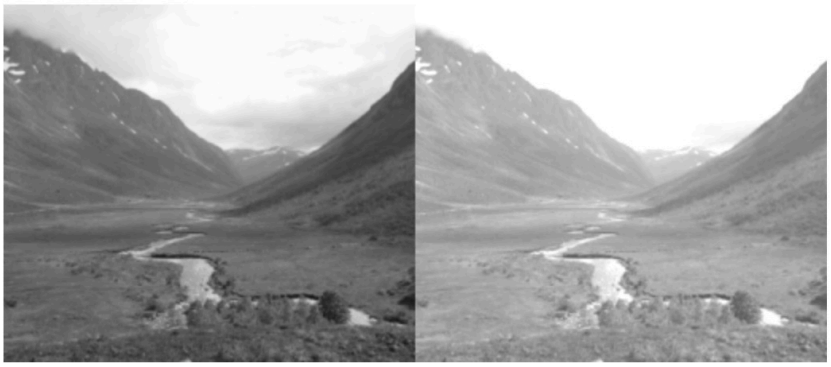

Figure 2.7-1 Brightness adjustment. [14]

2.7.2.2 Contrast adjustment

Contrast is the difference between the maximum and the minimum pixel intensity of an image. To change the contrast of an image, we change the range of the luminance value presented in the input pixels. It is mathematically suggested by the following function:

$$g(i,j)= \alpha\, f(i,j)$$

Equation 2.7-2

where f(i,j) is the pixel located in the i-th row and j-th column of the input image, g(i,j) is the pixel located in the i-th row and j-th column of the output image and α is a weight parameter which is used to control the image contrast. See the discussion of histogram adjustment in Section 2.1.2 for more information.

Figure 2.7-2 Contrast adjustment. [14]

We can combine the brightness and contrast control in a single operation. It is mathematically represented as:

$$g(i,j)= \alpha\, f(i,j) + \beta$$

Equation 2.7-3

2.7.2.3 Scaling

The purpose of scaling is to generate a new image with different resolution. Details can be found in upsampling and downsampling methods in section 2.1.2.

Artificially enlarging the dataset using the combination of these three forms of image translation in image pre-processing has two extra benefits. Firstly, it allows the network to "look at" the seed image at different perspectives. Secondly, it allows the CNN model to work properly, because CNN often requires to be fed with a large dataset. Notice that data augmentation is different from *image pre-processing*. Image pre-processing processes the images so that they all meet some pre-defined standards whereas data augmentation artificially enlarges the dataset by applying random transformations to each pre-processed image.

2.8 Data post-processing

Combination of models is a very effective way to reduce testing errors. At the same time, it significantly increases the number of networks parameters and thus computers processing speed. Dropout is a very efficient method for simmulating model combination. Dropout sets the output of each hidden layer of a network to zero with the probability of 0.5. Propagation and back propagation processes will not consider the 'dropout neurons' into their calculations. The described process is repeated several times. At each time, a different set of outputs from hidden layers are set to zero. The network's prediction of a tuple is remains the same regarding the change of the dropout neurons. By performing these presented steps, dropout can simulate the combination of several neural networks.

When dropout is used, it will take more time for a neuron to learn more robust features in order to make predictions, since it can no longer rely on the existence of other neurons. Therefore, it will take more time for the network to converge.

2.9 GPU Concepts

The enormous amount of input data required by deep learning may considerably reduce computation time. This problem can be overcome by spreading and training the networks across processing units. There are two types of processing units: central processing unit

(CPU) and graphics processing unit (GPUs).

In comparison to CPU, GPUs are far more powerful and efficient in parallel computing. They can be used to train far larger training sets in considerably less time. Current GPUs support cross-GPU parallelization that allows GPU to read and write into one another's memory directly. Even though each GPU has limited memory, which may restrict the size of networks to be trained using one, a GPU can still distribute the network's size across multiple units.

Recent deep learning toolkits are mostly developed based on CUDA library. This library only supports NVIDIA GPU card with *compute capability* >= 3.0. Compute capacity is a number that defines general specifications and available features of a GPU card. Details about general specifications and available features associated with each compute capacity value can be found in CUDA toolkit documentation [24].

2.10 CNN Software Frameworks

Framework	Base language	API	Single GPU execution speed	Ready-to-use low-level operators for writing new models	GPU memory for training large models	Ease to use
Tensorflow	Python and C++	- Python and C/C++ - Mathematical operations are supported with numpy	Slower than other frameworks	Fairly good	Not so good	- Sample code and tutorials available
Caffe	C++, Python	C++, command line, Python, MATLAB	Slower than other frameworks	Fairly good	Better than Torch	- Sample code and to tutorials are somewhat confusing because different versions are developed by different people - Code might not need to be written to train models
Theano	Python	- Python - Mathematical operations are supported with numpy	Faster than Caffe and Tensorflow	Many basic operations	Great	- Sample code and tutorials available - Error messages are difficult to understand, therefore can be unhelpful
MXNET	C++, Python, Julia, Matlab, R, Scala	C++, Python, Julia, Matlab, JavaScript, R, Scala	Faster than Caffe and Tensorflow	Very few	Excellent	- Sample code and to tutorials are somewhat confusing because different versions are developed by different people
Torch	Lua	Lua	Faster than Caffe and Tensorflow	Many basic operations	Fair	- Easy to set up - Error messages are helpful

Table 2.10-1 Comparison among different frameworks

Table 2.10-1 [21,22,23] shows a comparison among different frameworks. All of the frameworks in the table support parallelism through GPU mode using CUDA library. They also have an ability to produce a series of matrix operation (computation graph) on which automatic differentiation can be performed. With automatic differentiation, one does not have to hand-code a new variation of back propagation each time a new arrangement of the CNN is made.

Each framework, however, has some tradeoffs that we will have to consider before making a decision on which one to use. Theano supports the numpy library for mathematical operations, provides considerably fast single GPU speed, great GPU memory support for training large models and many basic operations. However, its error messages are difficult to understand. MXNET is the best in terms of GPU memory. However, it reportedly has confusing tutorials and a poor set of ready-to-use low-level operators. Torch has the best ease of use, many basic operations and compromising GPU speed and memory, but it uses the less well-known Lua programming language as its API. Caffe and Tensorflow are the most widely use frameworks recently. Caffe allows us to train models with little modification to its existing models. Nevertheless, it has supposedly confusing tutorials, slower GPU speed than other frameworks and it does not have numpy support for mathematical operations. Tensorflow does not have very good GPU memory and GPU speed but it has a fairly good set of ready-to-use low-level operators and sample tutorials. For this project, we consider experimenting Caffe and Tensorflow as they have a balanced compromise among different aspects and appear to be adopted by most developers.

2.11 Related research

2.11.1 Researches using CNN to classify images

CNN has proved to be very efficient in object recognition. A. Krizhesy et al. (2012) successfully modeled a deep CNN "to classify the 1.2 million high-resolution images in the ImageNet […] into the 1000 different classes" with "top-1 and top-5 error rates of 37.5% and 17.0%". The networks contained five convolutional layers, three max pooling layers, three fully connected layers and a 1000-way softmax. Data argumentation, overlap pooling and dropout were used to reduce data overfitting. To improve the processing speed, the networks were split into different parts, which were trained on multiple GPUs [1].

Hokuto et al. (2014) developed a food detection and recognition deep CNN trained from 20,000 samples of food items. The networks consisted of two convolutional layers, one ReLU layer and was able to detect up to 93.8% and recognize up to 72.39% of testing food items [5].

2.11.2 Variations in algorithms in CNN used for classifying fine-grained objects

Krizhevsky et al. (2012) has taken a step further to the traditional pooling. Suppose a pooling layer consists of a grid of pooling units, each summarizes a neighborhood of size z x z and stride s. Traditionally, we set z = s, where we obtained non overlap pooling. Now we will set s < z to obtain overlap pooling. A. Krizhevsky et al. has pointed out this is a more effective pooling technique as it reduces the top-1 error rate by 0.4% and top-5 error rate by 0.3% [1].

Jonathan et al. (2014) exploited deep CNN in fine-grained object recognition. The objective was to identify different car models. The networks were built by adapting the model from A. Krizhesy et al. with little variation. A deep CNN model consists of two convolutional layers, three fully connected layers and a softmax loss were used to extract useful features from the seed image. The seed image then is used to retrieved its nearest neighbors (those that has the same pose with it). Only parts with highest energy detected from this seed image and its neighbors are chosen because they are likely to be important

when describing the seed image. Learnt features obtained earlier from CNN are then pooled in the regions of each selected part. Those regions are said to describe critical parts for the class where seed image belongs. This networks were able to categorize testing data with accuracy up to 73.9%. However, the described technique is only useful for recognizing pictures with the same pose [4].

Aäron et al (2015) attempted to build a CNN to classify grayscale 30000 images of plankton into one of 121 classes. Input images were preprocessed in by downsampling. 300 CNN models were trained and several of them were combined to improve the final accuracy. A significant variation made when training the model was to exploit the viewpoints in pooling. One example of the models that works well consisted of 10 convolutional layers, 4 pooling layers and 3 fully connected layers. Models were trained on the NVIDA GPUs using various overfitting-reducing techniques: data argumentation, ReLU, dropout and overlap pooling. The best models had an accuracy of 82% on the testing set and top 5% accuracy of 98% [3].

Sander et al. (2015) built a deep CNN to measure approximately 900,000 galaxy morphology by exploiting viewpoints in pooling. After preprocessing and data augmentation, viewpoints are extracted by applying the combination of flipping, rotating, cropping to the input image. Every viewpoint is then presented to the same convolutional architecture in a separated path. The resulting feature maps from each viewpoint are first concatenated in to a single vector, then processed by a set of fully connected layer to obtain predictions. The benefit of exploiting viewpoints in pooling is that it allows the network to "look at" the image at different angles. The model was claimed to "reproduce their consensus with near-perfect accuracy (> 99%) for most questions" in Galaxy Challenge Contest 2015. Techniques discussed in A. Krizhesy et al. were also applied to reduce data overfitting [2].

Chapter 3

Design and Methodology

In this section, we will describe our approach to developing a valid model for sugarcane grading using ground-level images. We will first discuss the experimental set up. We will then discuss the successive processing steps used to obtain a set of predictive probabilities for each input image. The successive steps consist of the following: data preprocessing, data argumentation, view point extraction, convolutional neural network experimental design and model evaluation.

3.1 Experimental set up

3.1.1 Gathering data

The dataset consists of sugarcane field images taken from ground level and is provided by Mitrphol company. As suggested by other research, a minimum amount of 1000 pre-labeled samples in the dataset is required in order to train the convolutional neural networks. For the time being, we are provided with 31 pre-labeled image examples, which are divided into 4 growing seasons of sugarcane. We will use them to develop some simple models so as to evaluate the use of different frameworks. The full dataset is expected to be delivered by the end of the year.

3.1.2 Plan on experimenting different frameworks

As mentioned in Section 2.10, we consider experimenting Caffe and Tensorflow as they have a balanced compromise among different aspects and appear to be adopted by most developers. In this section, we will discuss our plan to study the use of the two aforementioned frameworks. Then we will evaluate the usability of each framework based on the following criterions:

The ease of the framework installation: We can evaluate the ease of installation by answering the following questions. Is the documentation for installing the framework easy to follow? Does the framework requires many other toolkits to be installed together with it? And if yes, is the installation of the supporting toolkits are complicated?

The ease of following the framework documentation: The documentation must provide instructions that are easy to follow. It must also contain understandable explanations of all operations that the framework provides.

The ease of constructing the network: A good framework should provide a good set of ready-to-use low-level operators for writing new models. It should also allow us to modify the networks' structure easily. Finally, it must support parallelism through GPU mode, which will allow us to train the networks with a reasonable amount of time.

The availability and accessibility of sample code and tutorials: Sample code and tutorials are important for us to understand how the framework works. If there are too little sample code and tutorials available, we might have difficulty in learning and modifying code in the framework.

The ease of understanding error messages from the framework: Error messages have to be helpful and easy to understand.

3.1.2.1 Tensorflow

3.1.2.1.1 Installation

Before installing Tensorflow, we will need to set up a few things. First, we will need to install Python. Details on how to install Python can be found in Python documentation [28]. Next, we will need to download and install CUDA toolkit in order to use Tensorflow CUDA version. Details can be found in CUDA installation documentation [29]. Finally, we can

install Tensorflow on the machine. Different ways of installing Tensorflow is supported. In our case, we will try to install it via Virtualenv with GPU enabled mode [26], because it allows us to install TensorFlow in its own directory without affecting any existing Python programs on our machine.

3.1.2.1.2 Testing plan

In order to evaluate this framework, we plan to refine the code from Tensorflow documentation on 'how to recognize hand-written digits from digital images in the MNIST data-set' to build a simple model that suits our problem. The code refinement are as follows. First, the example code attempted to read images and labels of binary format, which is not our case. We will need to read all images in the dataset and associate each with a label. Details on how we did this will be explained in Section 3.2.

We will reuse the next parts of the example code, with some modifications to fit the refinement we made at the beginning. Each time the system read an image, it will perform on this image a set of random transformations to create n new distorted images. The distorted images will inherit their labels from the original image. The original image and the distorted images will then be stored in a batch together with their labels. There is a finite number of batches, each of which stores a finite number of training images. The CNN is initialized with random weights and bias. At each iteration, a new batch is selected.

However, at this point, we want to try extracting a set of view points from each image in the batch and present each view point to the same convolutional architecture in a separated path, as suggested by Sander et al. (2015) [2]. Since the example code did not attempt do this, we need to modify it to fit our case. For each image in the selected batch, we will extract a set of view points and put them in an another, separated batch. We will

then present this new batch to the same convolutional architecture, but we will concatenate the results from the last ReLU layers of all convolutional architecture before sending them to the fully connected layers, where each input image is mapped with a suitable output class. Finally, Tensorflow provides a back-propagation method that allows all initial weights and bias in the networks to be updated so as to minimize the error rate within a single call.

We will assess the usability of the framework using the criterions specified earlier in this section based on how much trouble we will encounter during the experiment.

3.1.2.2 Caffe

3.1.2.2.1 Installation

Before installing Caffe, we will first need to install its dependencies. Caffe has some compulsory dependencies as follows. CUDA library is required to support GPU mode. BLAS library is required to support vector computation. Since Caffe base language is C++, Boost package is also required to support its C++ library. And Python is required to support its interface.

Besides these compulsory dependencies, Caffe has some optional dependencies. Caffe optional dependencies are OpenCV for image processing and cuDNN for GPU acceleration.

We will then install Caffe according to instructions on Caffe documentation [27].

3.1.2.2.2 Testing plan

In order to evaluate this framework, we plan to modify the imagenet tutorial in Caffe documentation to make the model trained on our sugarcane image.

The first step is to split our image dataset into two parts, one of which is the training set, the other is the testing set. There are three splitting strategies as described in Section 2.3.2. We will try the 'hold out' strategy because it is the most widely used.

After having the training and testing data all set, we will then refine the sample code so that it is able to read all images in each dataset and associate them with a label. To do this, we

will have to create a text file for each dataset. Each line in the text file is a directory to an image in the dataset and is followed by its label, as described in Section 3.2.

The next step is to inform the create_imagenet.sh file about the directory to the new dataset. To do this we will need to modify all arguments in that file that refers to one of the following: the training set directory, the testing set directory, the training set text file or the testing set text file. When we compile the create_imagenet.sh file, it will generate a training leveldb and testing leveldb directories. Caffe will work with these leveldb directories instead.

We will then need to inform make_imagenet_mean.sh about our new leveldb directories. The file make_imagenet_mean.sh when being compile will create a file name imagenet_mean.binaryproto. Caffe uses this file to subtract the image mean value from each image in order to normalize our inputs. This step is probably irrelevant to our problem since we want to mimic the photos taken in different conditions with different mobile phones. Therefore, we may want to find a way to set the image mean value to zero.

The final step is to notify each of these following file: solver.prototxt, train_val.prototxt and deploy.prototxt on the newly created leveldb directories and imagenet_mean.binaryproto.

The file solver.prototxt keeps information about how we will use our dataset to train and test the CNN. One example is the number of iteration for the propagation and back-propagation cycle. The train_val.prototxt is where data argumentation is performed. The file deploy.prototxt defines the network structure. This is where the organisation of convolutional, pooling, ReLU and fully connected layers are defined. We plan to play around with this file in order to see how easily can we modify the networks' structure using this framework.

We will assess the usability of the framework using the criterions specified earlier in this section based on how much trouble we will encounter during the experiment. Then we will conclude which framework is better for our experiment.

3.1.3 Computer resource

We need a powerful GPU card that supports CUDA library to train the CNN. CUDA library documentation recommends any NVIDIA GPU card with compute capacity greater than 2, but the most powerful are some models of the NVIDIA Tesla and any model of the NVIDIA Geforce Titan [24]. We plan to use the KMUTT innosoft high performance computing service to train our CNN [25]. The innosoft server hosts a NVIDIA Tesla K10 card. This card has a compute capacity of 3.0 and is thus suitable for the experiment.

Tensorflow provides tf.device method and Caffe provides caffe.set_device(gpu_id) method. Both of these methods allow us to specify the remote GPU server that we would like to train our networks on.

3.2 Data preprocessing

First, we will make sure that every image is associated with a label and all labels can be automatically interpreted by our system. For the moment, we have not obtained the full dataset. Therefore, it is difficult to tell exactly how we will let our system handle the labels.

In the example dataset we currently have, images are stored in the local machine and each of them is labeled by its name. We can handle this example dataset as follows. We will first label each file with a number in the range [0,2] according to its name, with 0 being **poor**, 1 being **medium** and 2 being **good**. We will then keep all information of the images in the dataset in a text file, where each line is a directory to an image followed by its label. Our networks can look at this file in order to locate each image and identify its label. Finally, if we look into the output layer of the CNN, we will see this layer contains three output neurons, each represents a class. Our final step is thus to encode decimal labels into binary labels where each digit of a binary label represents a class value. Nevertheless, the complete dataset could be split into different folders and labeled by folder names, in which

case we could read each folder name and impose the label to all images contained within each folder. It could be hosted by a remote server, in which case the approach is the same with one variation: all directories are directories of files and images on the server.

Next, we will need to standardize each experimental image before sending them to the input layer of the CNN. Having all images standardized allows us to process them easier. We will standardize the images as follows. We will first rotate images that are upside down. We will then have resized each experimental image into a squared shape. Resizing images speeds up the training process with little or no effect on the accuracy [2]. Both Tensorflow and Caffe provide functions to perform these operations easily.

3.3 Data argumentation

Due to the limitation of the training set size, we will need to artificially increase the number of the training samples. Each training sample is randomly distorted into n forms. However, in order to mimic the photos taken in different conditions with different mobile phones, they cannot be extremely distorted.

We will experiment the network with different value of n and see whether adjusting this value will have any significant effect on the accuracy.

Each distorted method is a combination of the following elementary forms of transformations. Details about the mechanic behind each form of transformation was discussed in Section 2.10.

3.3.1.1 Brightness and contrast adjustment

We plan to test the brightness and contrast adjustment operations with different input value and print out the resulting image. By doing this, we can look for the range in which the resulting image is most similar to images taken by mobile phones. We will then able to choose a random value within this range an impose it to different input images.

3.3.1.2 Cropping

We will randomly crop an area in the central region of an image for two reasons. Randomly cropping mimics photos of the same object taken by different people, while cropping an area in the central region retains the important information from an image. We plan to crop out from 20-40% and retain from 60-80% the area in the central region of an image.

Tensorflow provides operations that allow us to transform an image within a single call without writing extra code. The operations that we plan to use are:

- tf.image.adjust_brightness(image, delta) for adjusting an image brightness
- tf.image.adjust_contrast(image, delta) for adjusting an image contrast
- tf.image.central_crop(image, central_fraction) for cropping an image
- Details on how to use these operations can be found on Tensorflow documentations [26].

Caffe provides the **transform_param**, where we can set a value of the **contrast_adjustment**, **brightness__adjustment** and **crop** variables to make it do the same job. However, depending on which Caffe version is used, we may have to add some C++ code to the transform_param function in order to specify the extract job for each operation.

3.4 View points extraction

After data argumentation, we will extract an n different view points by cropping, adjusting the brightness and contrast of each input image. By theory, view points extraction allows the CNN to ‘look at’ each input image at different perspective. Thus, exploiting view points extraction in constructing a model should improve it accuracy. To test this theory, we

will build our models both with and without view points. We will also try to increase n in order to see whether it will increase the accuracy.

3.5 Experimental designs

3.5.1.1 Strategy for organizing training and testing set

Before building any model, we will need to divide the given dataset into training and testing sets. We will use the 'hold out' strategy because it is the most popular.

3.5.1.2 Data segregation strategy

It is uncertain whether models trained with datasets segregated by different growing seasons will have better accuracy than a single model trained with the entire dataset. We will begin by building one model for the entire dataset. However, if we have a big enough amount of data on each growing seasons and if time allows us, we will develop a model for each season.

3.5.1.3 Networks architecture

After choosing an appropriate framework, we will begin to experiment different CNN architectures. We will start with the CNN architecture suggested by the classic article by Krizvevsky et al. (2012). It consists of 8 trainable layers. There are 5 convolutional layers, all of them have squared filters with size 11,5,3,3,3 respectively. The first, second and fifth convolutional layers are followed by an overlap 3 by 3 max pooling layer with stride 2. The feature maps from the last max pooling layer is then processed by three fully connected layers. The results from the final fully connected layer is fed to a softmax function, which will produce a prediction over the three classes. [1]

Next, we will modify this architecture by adjusting different levels of independent variables and see whether any of these adjustments will help increase the accuracy.

The simplest adjustment is the size of input images. Krizvevsky et al. (2012) uses squared input images of size 224. Larger input images would cost the CNN more training time. In our case, we would like to know the input images size that generates the best accuracy with an optimal amount of time.

The size of the filters and the use of overlap pooling are probably the two least time consuming adjustment because they do not considerably expand the size of a model. However, previous experiments showed that it is not always the case that they will affect the accuracy. In this experiment, we will first test whether the use of overlap pooling will increase the accuracy. We will then adjust each filter size up to 2 by 2 pixels difference from the original to test whether they have some significant effects to the accuracy.

According to Aaron et al. (2015), view point extraction would increase the accuracy. Nevertheless, view points extraction increase the number of input images and the number of CNNs by n times, where n is the number of view points extracted [2]. We would like to experiment view point extraction on our problem to see whether it will improve the prediction capability of a model. We will start by extracting two view points and gradually increase it according to the available time.

Increasing the number of layers of each type would potentially increase the accuracy. However, it will take more time to train a CNN with more layers. We will gradually increase the number of convolutional layers according to the time we have.

3.5.1.4 Results evaluation

In order to evaluate the results, we will calculate the accuracy and the error rate of each model. We will compare the predictions made by a model with the original label of each input image. The accuracy is the percentage of correct prediction over the total number of input images. The error rate is the negation of the accuracy.

For debugging purpose, we will print out both the input label and the label predicted by a model for each input images. An input label and a predicted label at any corresponding position will refer to the same input image.

Below is an example of the labels for a batch of three images.

[0 0 1
 1 0 0
 0 1 0]

Each line contains labelling information for one image. Each digit in a line represents one class. The first digit represents class ‘poor’, the second represents class ‘medium’ and the last represents class ‘good’. When the digit is ‘1’, the class that it refers to is active, otherwise it is passive.

Reference

[1] A. Krizhevsky, I. Sutskever, G.E. Hinton. ImageNet classification with deep convolutional neural networks. In Advances in Neural Information Processing Systems 25 (NIPS'2012), 2012

[2] Sander Dieleman, Kyle W. Willett and Joni Dambre.Rotation-invariant convolutional neural networks for galaxy morphology prediction. Mon. Not. R. Astron. Soc. 000, 1–20 (2014).

[3] Aäron van den Oord, Ira Korshunova, Jeroen Burms, Jonas Degrave, Lionel Pigou, Pieter Buteneers. Classifying plankton with deep neural networks. First prize of The National Data Science Bowl competition, March 2015.

[4] Jonathan Krause, Timnit Gebru, Jia Deng, Li-Jia Li, Li Fei-Fei. ICPR, 2014, supported by an ONR MURI grant and the Yahoo! FREP program. Learning Features and Parts for Fine-Grained Recognition.

[5] Hokuto Kagaya, Kiyoharu Aizawa, Makoto Ogawa. MM'14, November 3–7, 2014. Food Detection and Recognition Using Convolutional Neural Network

[6] Evgeny A. Smirnov*, Denis M. Timoshenko, Serge N. Andrianov. Comparison of Regularization Methods for ImageNet Classification with Deep Convolutional Neural Networks. 2013 2nd AASRI Conference on Computational Intelligence and Bioinformatics

[7] Yoshua Bengio, Ian Goodfellow, and Aaron Courville. Draft book in preparation. Deep learning, Jan 2016

[8] Jiawei Han & Micheline Kamber. Data Mining Concepts and Techniques, 3rd edition, 2012

[9] Jonathan Sachs. 1996-1999. Digital Light & Color

[10] Rafael C. Gonzalez, Paul Wintz. Digital Image Processing, 2nd edition, 1987

[11] Wikipedia. RGB color model

[12] Stanford University. Convolutional neural networks for visual recognition

[13] Mathworks. Convolutional neural networks documentation

[14] Pippin online tutorial

[15] Grzegorzgwardys. Convolutional Neural Networks

[16] Eindhoven University of Technology – PARsE. Convolutional Neural Networks

[17] Matthew D. Zeiler, Rob Fergus. Nov 2013. Visualizing and Understanding Convolutional Networks.

[18] Dominik Scherer, Andreas Mu¨ller, and Sven Behnke. 20th International Conference on Artificial Neural Networks (ICANN), Thessaloniki, Greece, September 2010. Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition.

[19] Matt Mazur. A Step by Step Backpropagation Example. https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/. Accessed 7 Oct 2016.

[20] Bruna J., A. Szlam and LeCun Y. 27 Feb 2014. Signal recovery from Pooling Representations.

[21] Trivedi A., Microsoft Data Scientist. Deep Learning Part 1: Comparison of Symbolic Deep Learning Frameworks. http://blog.revolutionanalytics.com/2016/08/deep-learning-part-1.html. Accessed 7 Oct 2016.

[22] DL4J vs. Torch vs. Theano vs. Caffe vs. TensorFlow. http://deeplearning4j.org/compare-dl4j-torch7-pylearn.html#caffe. Accessed 7 Oct 2016.

[23] Murphy J. Deep Learning Frameworks: A Survey of TensorFlow, Torch, Theano, Caffe,

Neon, and the IBM Machine Learning Stack. https://www.microway.com/hpc-tech-tips/deep-learning-frameworks-survey-tensorflow-torch-theano-caffe-neon-ibm-machine-learning-stack/. Accessed 7 Oct 2016.
[24] CUDA toolkit documentation. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capability-6-x. Accessed 12 Oct 2016.
[25] KMUTT innosoft high performance computing service. http://hcp.innosoft.kmutt.ac.th/site. Accessed 16 Oct 2016.
[26] Tensorflow documentation. Accessed 16 Oct 2016.
[27] Caffe documentation. Accessed 16 Oct 2016.
[28] Python documentation. Accessed 16 Oct 2016.
[29] CUDA installation documentation. Accessed 16 Oct 2016.