DEEP LEARNING FOR SATELLITE IMAGE ANALYSIS

by

Lior Bragilevsky

Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Honours Bachelor of Applied Science in the School of Engineering Science Faculty of Applied Science

© Lior Bragilevsky 2017 SIMON FRASER UNIVERSITY Fall 2017

Copyright in this work rests with the author. Please ensure that any reproduction or re-use is done in accordance with the relevant national copyright legislation.
APPROVAL

Name: Lior Bragilevsky
Degree: Bachelor of Applied Science
Title of Thesis: Deep Learning for Satellite Image Analysis

Dr. Glenn Chapman, P.Eng.
Director
School of Engineering Science, SFU

Examinining Committee:

Chair and Academic Supervisor:
Dr. Ivan Bajić, P.Eng.
Professor
School of Engineering Science, SFU

Committee Member:
Dr. Parvaneh Saeedi, P.Eng.
Associate Professor
School of Engineering Science, SFU

Committee Member:
Dr. Bernhard Rabus
Professor
School of Engineering Science, SFU

Date Approved: DECEMBER 6, 2017
Abstract

Deep learning architectures have the potential of saving the world from losing football field-sized forest areas each second. These architectures possess large learning capacities when compared to conventional machine learning architectures, and thus are trained on sizable data-sets to efficiently extract both coarse and fine features from various image scenes. As a result, they can provide crucial information that is needed to manage the deforestation process and its consequences on the environment and ecosystem more effectively.

This thesis outlines the two deep learning based systems designed for satellite image analysis. The first system analyzed satellite images of the Amazon, and the goal was to interpret the image content by providing a set of labels that best describe it. The highest performing architecture was able to achieve a score of 92.886% while a combination of several high performance, yet uncorrelated, architectures increased the overall score to 93.070%. This result is only 0.248% lower than what current state of the art algorithms achieved on the same task. The second system was designed to detect the presence of clouds in Landsat 8 images by analyzing small chips within each large image. This system produced cloud masks, which were then compared to the corresponding ground truth cloud masks obtained from the provided images. The predicted cloud masks were able to achieve an average score of 92.931%, which is very high for the given accuracy measure.
After months of hard work and dedication, I managed to complete my undergraduate thesis at last. Now, I would like to take the opportunity to mention and sincerely thank everyone who has supported me throughout this journey.

Foremost, I would like to express my gratitude to my academic supervisor, Dr. Ivan Bajić, for being patient, understanding, motivating, and supportive throughout the entire process of my thesis documentation. Each meeting presented new ideas and learning opportunities, which were essential to the successful completion of this thesis. Most importantly, thank you for inspiring me to pursue graduate studies and a career in machine learning research.

To my thesis committee members, Dr. Parvaneh Saeedi and Dr. Bernhard Rabus, thank you for the valuable constructive criticism you provided on both the thesis proposal and this thesis. I would also like to mention that I was really motivated by the genuine interest in my research shown from both sides.

To my Landsat 8 image analysis system project partner, Thomas Krammer, I sincerely thank you for your valuable teamwork, explanations, understanding, and support with the pre-processing stage.

To my laboratory teammates, Dr. Stephen Makonin, Hyomin Choi, Saeed Ranjbar Alvar, Herath Gedara Chinthaka Pathum Dinesh, and James Lin, thank you for the machine learning related ideas, scientific documentation suggestions, and general support/encouragement.

Lastly, to my parents, Anna and Igor Bragilevsky, and brother, Dan Bragilevsky, thank you for the continuous support, praise, and encouragement throughout my entire academic and research career.
Table of Contents

Approval ii

Abstract iii

Acknowledgements iv

Table of Contents v

List of Tables viii

List of Figures ix

1 Introduction 1

1.1 Deep Learning Overview 2

1.1.1 Backpropagation 3

1.1.2 Neural Network Challenges 4

1.2 Convolutional Neural Networks (CNN) 4

1.2.1 Convolutional Layer 4

1.2.2 Activation Layer 6

1.2.3 Pooling Layer 6

1.2.4 Classification Stage 7

2 Amazon Image Analysis System 9

2.1 Amazon Satellite Images 9

2.1.1 Class Labels 10

2.1.2 Accuracy Measure 10

2.2 Convolutional Neural Network Architectures 11

2.2.1 Custom CNN Architecture 12

2.2.2 Pre-trained CNN Architectures 14

2.3 Training Procedure 15

2.3.1 Data Augmentation 16

2.3.2 Optimal Threshold Selection 17

2.4 Post Processing 18
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4.1</td>
<td>Weighted Majority Voting Ensemble</td>
<td>18</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Weighted Soft Voting Ensemble</td>
<td>20</td>
</tr>
<tr>
<td>2.5</td>
<td>Results</td>
<td>22</td>
</tr>
</tbody>
</table>

3 Landsat 8 Image Analysis System | 24

3.1 Landsat 8 Images | 24

3.1.1 Class Labels | 26
3.1.2 Accuracy Measure | 26

3.2 CNN Binary Classifier | 27

3.3 Training Procedure | 27

3.3.1 Data Augmentation | 28
3.3.2 Full Resolution Image Based Threshold Selection | 29

3.4 Current Results | 29

3.4.1 Cloud Mask Generation | 29

4 Conclusion & Future Work | 35

References | 37

Appendix A Amazon Image Analysis System | 40

A.1 Load Required Modules | 40
A.2 Function Definitions | 41

A.2.1 Accuracy Measure | 41
A.2.2 Threshold Optimization | 41

A.3 Image & Label Preparation | 42
A.4 Custom CNN Architecture | 43
A.5 Pre-trained CNN architectures | 44
A.6 Model Training | 45
A.7 Model Testing - Label Predictions | 47
A.8 Ensemble Procedures | 49

A.8.1 Weighted Majority Voting Ensemble | 49
A.8.2 Soft Weighted Voting Ensemble | 50

Appendix B Landsat 8 Image Analysis System | 52

B.1 Additional Modules Required | 52
B.2 Jaccard Index of Similarity | 53
B.3 Threshold Selection | 54
B.4 Data-set Preparation | 54
B.5 CNN Binary Classifier | 55
B.6 Training Stage | 56
B.7 Test Stage | 58
List of Tables

Table 2.1 Satellite Image Specifications ........................................... 9
Table 2.2 Image Labels with Corresponding Frequency and Index Values .... 11
Table 2.3 $F_2$ Scores of Various Architectures ..................................... 14
Table 2.4 ResNet50 Optimal Threshold Values ....................................... 18
Table 2.5 $F_2$ Scores of Ensemble Methods ........................................... 23
Table 2.6 Training Data-set Label $F_2$ vs. Jaccard Index Scores ............... 23

Table 3.1 Landsat 8 Image Channels Utilized ........................................ 24
Table 3.2 Occurrence Frequency and Index Values of Chip Labels .............. 26
Table 3.3 CNN Binary Classifier Segmentation Performance .................... 29
Table 3.4 Binary Classifier Cloud Masks ............................................. 30
Table 3.5 Raw Cloud Masks & Confusion Matrix .................................... 32
Table 3.6 Cloud Mask Jaccard Index Scores ......................................... 33

Table B.1 Extra Details for Landsat 8 Image Channels ............................ 63
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>Deep Learning Performance</td>
<td>1</td>
</tr>
<tr>
<td>Figure 1.2</td>
<td>Neural Network Architecture</td>
<td>2</td>
</tr>
<tr>
<td>Figure 1.3</td>
<td>Loss Function Optimizer (Stochastic Gradient Descent)</td>
<td>3</td>
</tr>
<tr>
<td>Figure 1.4</td>
<td>Backpropagation in Neural Networks</td>
<td>3</td>
</tr>
<tr>
<td>Figure 1.5</td>
<td>Local Receptive Field of Convolutional Layer</td>
<td>5</td>
</tr>
<tr>
<td>Figure 1.6</td>
<td>Feature Maps from Convolutional Layer</td>
<td>5</td>
</tr>
<tr>
<td>Figure 1.7</td>
<td>ReLU &amp; Sigmoid Activation Functions</td>
<td>6</td>
</tr>
<tr>
<td>Figure 1.8</td>
<td>Local Receptive Field of Pooling Layer</td>
<td>6</td>
</tr>
<tr>
<td>Figure 1.9</td>
<td>Feature Maps from Pooling Layer</td>
<td>7</td>
</tr>
<tr>
<td>Figure 1.10</td>
<td>Feature Vector Generation</td>
<td>7</td>
</tr>
<tr>
<td>Figure 1.11</td>
<td>CNN Classification Block</td>
<td>8</td>
</tr>
<tr>
<td>Figure 2.1</td>
<td>Sample Satellite JPEG (top) and GeoTIFF (bottom) Images</td>
<td>10</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>Custom CNN Architecture</td>
<td>13</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>Pre-trained CNN Architecture Design</td>
<td>14</td>
</tr>
<tr>
<td>Figure 2.4</td>
<td>ResNet50 Training Stage Performance Log</td>
<td>15</td>
</tr>
<tr>
<td>Figure 2.5</td>
<td>Effects of Label Distribution Bias on Training Stage</td>
<td>16</td>
</tr>
<tr>
<td>Figure 2.6</td>
<td>Reflection for Filling Gaps Formed by Rotation</td>
<td>17</td>
</tr>
<tr>
<td>Figure 2.7</td>
<td>Weighted Majority Voting Ensemble</td>
<td>20</td>
</tr>
<tr>
<td>Figure 2.8</td>
<td>Weighted Soft Voting Ensemble</td>
<td>22</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>Sample Near-infrared Channel Landsat 8 Images</td>
<td>25</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Landsat 8 Custom CNN Training Stage Performance Log</td>
<td>28</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Confusion Matrix</td>
<td>32</td>
</tr>
<tr>
<td>Figure B.1</td>
<td>Landsat 8 Naming Conventions</td>
<td>63</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In the early 2000s, deep learning architectures gained popularity due to their ability to outperform conventional machine learning models when the size of the data-set is large, as shown in Figure 1.1.

From Figure 1.1 it is clear that the prediction accuracy of deep learning models continues to increase, while the prediction accuracy of shallow machine learning models approaches its asymptotic maximum, as the size of the training data-sets increase. This key difference comes from the fact that deep learning models contain significantly more layers than machine learning models, allowing them to possess a much larger learning potential. This property of deep learning models is particularly useful for satellite image analysis applications and can be exploited to extract the crucial information needed to better understand and prevent deforestation around the world.

This thesis research focused on designing and applying deep learning models for satellite image analysis. Chapter 1 provides a brief overview of deep learning, which focuses on
the properties of deep neural networks that were used in this work. Chapter 2 introduces the different deep learning models and ensemble techniques [2] utilized to classify satellite images of the Amazon basin. Chapter 3 describes the Landsat 8 Imagery System designed to detect clouds in satellite images. Lastly, chapter 4 summarizes the results obtained with the satellite image analysis systems discussed in chapters 2 & 3 and presents possible future applications for these systems.

1.1 Deep Learning Overview

As their name suggests, deep learning models often contain significantly more layers than conventional machine learning models. These layers are made up of nodes, called neurons, forming what is known as a neural network [1]. For any given layer of a neural network, its neurons are connected to neighbouring layers, with the connection strength being indicated by weights. The general structure of a neural network is shown in Figure 1.2.

![Figure 1.2: Neural Network Architecture](image)

As shown in Figure 1.2 above, generally a labelled input (supervised learning) is provided to the input layer, which produces weights ($w_i$) that connect its neurons to the neurons of the first hidden layer. This hidden layer also generates weights ($w_j$) which connect its neurons to the following hidden layer. This process continues until the neurons of the final hidden layer get connected to the output layer with their corresponding weights. During the training stage these weights are continuously updated, using the backpropagation method described next, to determine the optimal weights. During the test stage the neural network generates predictions, which should closely match the desired target output if the network trained on the provided input and learned properly.

2
1.1.1 Backpropagation

During the training stage, the weights of each neuron are adjusted such that the neural network minimizes its prediction error, which is achieved by the use of a loss function. This loss function is minimized during training by an optimization algorithm, known as an optimizer, to hopefully find its absolute minimum point, as shown in Figure 1.3. An algorithm known as Stochastic Gradient Descent [3] is a common optimizer for neural networks, with many other optimizers being derived from it.

As shown in Figure 1.3, the rate at which the optimizer converges depends on the chosen initialization state. The optimizer uses a crucial concept, known as backpropagation (BP), to recursively adjust the weights of the neural network [1, 4]. Figure 1.4 shows the BP procedure for neural networks.
The BP procedure shown in Figure 1.4 consists of the following key steps:

1. The neural network generates predictions at the output layer from a provided input.
2. These predictions are compared with the target output to compute prediction errors.
3. Using BP, after each iteration (epoch) through the training data-set, neuron weights are adjusted to reduce the prediction errors.
4. Once the predefined stopping criteria are met, the training stage of the neural network ends.

1.1.2 Neural Network Challenges

A bottleneck of deep learning models is that they need to train/learn a very large number of parameters when compared to conventional machine learning models, making them capable of overfitting the training data-set [1]. Overfitting occurs when the neural network shows exceptional results on the training data-set, but fails to produce similar results, or generalize, on the test data-set. This was a major issue during the start of the deep learning era, making many researchers turn to conventional machine learning models such as Support Vector Machines (SVM), Decision Trees, and K-Nearest Neighbors [1].

With computational improvements, neural networks gained popularity as new regularization methods [5], such as dropout and data augmentation, were invented to prevent overfitting [1]. Dropout is used to prevent weight updates for a portion of neurons on the corresponding layer. Data augmentation increases the variety of the training data-set by adding random rotations, zooming, scaling, flipping, etc. These neural network improvements allowed deep learning models to outperform conventional machine learning models when trained on large scale data-sets.

1.2 Convolutional Neural Networks (CNN)

With the increased interest in deep learning models, Convolutional Neural Networks (CNN) were developed for object recognition, detection, and segmentation applications [1]. The CNN architecture consists of convolutional, activation, pooling, and classification blocks connected in a “feedforward” configuration, forcing information to propagate in the same direction from the input layer to the output layer.

1.2.1 Convolutional Layer

As seen previously, the neurons of a specific layer of a neural network are fully connected to the neurons of neighbouring layers. This is often not the case with CNN networks which have stages where the neurons of a hidden layer are only connected to a portion of the
previous layer, known as the local receptive field [6]. The local receptive field for a given convolutional layer is shown in Figure 1.5.

![Figure 1.5: Local Receptive Field of Convolutional Layer [6]](image)

The convolutional block creates many parallel hidden layers, known as feature maps [6]. The number of feature maps produced is equal to the number of convolutional filters applied. The creation of feature maps from a convolutional layer is illustrated in Figure 1.6.

![Figure 1.6: Feature Maps from Convolutional Layer [6]](image)

For the example shown in Figure 1.6 above, the convolutional layer used 5 filters of size $3 \times 3$ on the $32 \times 32$ input layer to produce 5 feature maps of size $30 \times 30$. This reduction in size is due to the convolutional layer “trimming” the edges of the input layer, as shown in Figure 1.5.
1.2.2 Activation Layer

Activation layers, also known as the non-linearity layers [1], are used to transform the values of the feature maps. Generally, the Rectified Linear Units (ReLU) activation function, shown in Figure 1.7a, is used in convolutional hidden layers to convert any negative value to 0 and linearly maps any positive value. Likewise, the Sigmoid activation function, shown in Figure 1.7b, is often used in the output layer to produce a probabilistic value bounded between 0 and 1. Thus, as information progresses through the model, large negative/positive values will be adjusted by the activation functions, allowing training to continue successfully.

![Figure 1.7: ReLU & Sigmoid Activation Functions](image)

1.2.3 Pooling Layer

Pooling layers are used to further reduce the size of the feature maps produced by the convolutional layers. This process filters out small values in the feature maps, making the output more noise resistant [1]. Figure 1.8 shows local receptive field of a given pooling layer.

![Figure 1.8: Local Receptive Field of Pooling Layer](image)
A pooling layer can also be applied to multiple feature maps as shown in Figure 1.9.

For the example shown in Figure 1.9, the size of the pooling window, known as the kernel, is $2 \times 2$. It is important to note that the step size (stride) of the kernels is chosen such that they do not overlap on a given feature map, as shown in Figure 1.8. This produces the same number of feature maps as in the convolutional layer, in this case 5, with a size of $15 \times 15$ each.

### 1.2.4 Classification Stage

Lastly, the classification stage generates the predicted outputs of the CNN architecture. Using the $N \times N$ feature map obtained from the pooling layer, a $1 \times N^2$ feature vector of neurons is created by transposing each row (starting at the top row) and appending the transposed rows to the end, as illustrated in Figure 1.10.
The neurons of the feature vector are then fully connected to the next layer, which in this case is the output layer, as shown in Figure 1.11.

![Figure 1.11: CNN Classification Block](image)

It is important to note that in Figure 1.11 the fully connected layer (FC), representing the feature vector, connects each of its neurons to every neuron in the output layer. Generally, the output layer contains less neurons than the feature vector ($M < N^2$).
Chapter 2

Amazon Image Analysis System

Every second around the world, football field-sized forest areas are lost due to deforestation, destroying the ecosystem and environment in the process. To better understand the deforestation process, large and otherwise inaccessible regions of the Amazon were monitored using satellites for many years [8, 9]. The first part of this thesis work focused on designing a deep learning based system to analyze images produced by these satellites. The designed architectures are able to detect subtle features in different image scenes, providing insight on how to manage the deforestation process more effectively. The contents of this chapter have been presented in [10].

2.1 Amazon Satellite Images

For this thesis work, the analyzed satellite images [11] came in two different formats, JPEG and GeoTIFF (Geo-referenced TIFF), of resolution $256 \times 256$. Relevant information regarding each format is summarized in Table 2.1.

<table>
<thead>
<tr>
<th>Format</th>
<th>Channels</th>
<th>Channel Type</th>
<th>Bits per Pixel</th>
<th>Maximum Pixel Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPEG</td>
<td>3</td>
<td>R, G, B</td>
<td>8</td>
<td>255</td>
</tr>
<tr>
<td>GeoTIFF</td>
<td>4</td>
<td>R, G, B, NIR</td>
<td>16</td>
<td>65535</td>
</tr>
</tbody>
</table>

In Table 2.1 above, the red, green, blue, and near infrared channels are represented by R, G, B, and NIR, respectively. Out of the 101,670 satellite images provided, 40,479 images were labelled and 61,191 images were not labelled. The labelled images were used as the training data-set from which the designed models learned. Whereas, the remaining unlabelled images made up the test data-set, which the designed models used to generate the label predictions.

Figure 2.1 shows a sample from the training data-set, where the top row displays the 3 channel JPEG images and the bottom row displays the corresponding 4 channel GeoTIFF images. To highlight the channel composition of each GeoTIFF image, they are displayed as
“data cubes,” where the top face is composed of the near infrared, red, and green channels, while the sides show the pixel magnitudes from each channel.

![Sample Satellite Images](image)

Figure 2.1: Sample Satellite JPEG (top) and GeoTIFF (bottom) Images

2.1.1 Class Labels

The task that this research focused on is considered a multi-label classification problem [12], as each satellite image may be described by up to 17 unique labels. On the following page, Table 2.2a provides a list of all the labels along with their occurrence frequency in the 40,479 training images.

Table 2.2a illustrates the bias present in the training data-set as the least frequent labels, such as ‘blow_down’ and ‘conventional_mine’, are each only associated with less than 0.25% of the training data. On the contrary, the label ‘primary’ is the most frequent label and is present in more than 90% of the training images.

2.1.2 Accuracy Measure

To determine how well any algorithm predicts the labels for input images, the $F_2$ accuracy score measure [13] was used. This accuracy measure was chosen to conform with the data-set providers [11], allowing the accuracy of the test data-set based predictions to be determined.

Refer to Appendix A.3 for a label assignment Python script
The general $F_\beta$ formula is computed using precision ($p$) and recall ($r$):

$$F_\beta = (1 + \beta^2) \cdot \frac{p \cdot r}{(\beta^2 \cdot p) + r}, \quad \therefore F_2 = \frac{5 \cdot p \cdot r}{4 \cdot p + r} \quad (2.1)$$

where

$$p = \frac{t_p}{t_p + f_p}, \quad r = \frac{t_p}{t_p + f_n} \rightarrow F_2 = \frac{t_p}{t_p + \frac{4}{5} f_n + \frac{1}{5} f_p} \quad (2.2)$$

and $t_p$, $f_p$, and $f_n$ represent the number of true positives, the number of false positives, and the number of false negatives, respectively (see Figure 3.3 for details). In Equations 2.1 & 2.2, recall represents the fraction of images for which each label was predicted, while precision indicates how accurate these predictions are.

### 2.2 Convolutional Neural Network Architectures

To tackle this multi-label classification challenge, a combination of a custom deep CNN architecture along with other pre-trained CNN architectures were implemented in Keras with Tensorflow backend. For each training image provided, its labels were converted into a 17-dimensional binary label vector whose indices serve as indicators for specific labels, as shown in Table 2.2b.

#### Table 2.2: Image Labels with Corresponding Frequency and Index Values

<table>
<thead>
<tr>
<th>(a) Labels &amp; Frequencies [11]</th>
<th>(b) Labels &amp; Index Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Label</strong></td>
<td><strong>Frequency</strong></td>
</tr>
<tr>
<td>primary</td>
<td>37840</td>
</tr>
<tr>
<td>clear</td>
<td>28203</td>
</tr>
<tr>
<td>agriculture</td>
<td>12338</td>
</tr>
<tr>
<td>road</td>
<td>8076</td>
</tr>
<tr>
<td>water</td>
<td>7262</td>
</tr>
<tr>
<td>partly_cloudy</td>
<td>7251</td>
</tr>
<tr>
<td>cultivation</td>
<td>4547</td>
</tr>
<tr>
<td>habitation</td>
<td>3662</td>
</tr>
<tr>
<td>haze</td>
<td>2695</td>
</tr>
<tr>
<td>cloudy</td>
<td>2330</td>
</tr>
<tr>
<td>bare_ground</td>
<td>859</td>
</tr>
<tr>
<td>selective_logging</td>
<td>340</td>
</tr>
<tr>
<td>artisinal_mine</td>
<td>339</td>
</tr>
<tr>
<td>blooming</td>
<td>332</td>
</tr>
<tr>
<td>slash_burn</td>
<td>209</td>
</tr>
<tr>
<td>blow_down</td>
<td>101</td>
</tr>
<tr>
<td>conventional_mine</td>
<td>100</td>
</tr>
</tbody>
</table>

Refer to Appendix A.2.1 for a Python algorithm outlining the $F_2$ score accuracy measure
For example, if the training image only contains the label ‘bare_ground’, its corresponding 17-dimensional binary label vector would be

\[ y = (0, 1, 0, 0, 0, ..., 0), \]

whereas, a training image which contains the labels ‘blow_down’ and ‘conventional_mine’ would have the binary label vector

\[ y = (1, 0, 1, 0, 0, ..., 0). \]

### 2.2.1 Custom CNN Architecture

The designed deep learning based custom CNN architecture, shown in Figure 2.2 on the next page, was inspired by the well known VGG network architecture [14]. Hence, it includes a sequence of Convolution-Convolution-Maxpooling (CCM) “super-layers,” each of which are made up of two back-to-back convolutional layers followed by a maximum pooling layer. In these super-layers, each convolutional layer has a ReLu activation function, and a specific number of \(3 \times 3\) filters. The depth of each layer indicates the number of filters applied to it, which increases towards the output of the architecture. The four CCM super-layers and their corresponding parameters are specified in the figure.

The output of the maximum pooling layer, belonging to the last super-layer, is fed to a classification block consisting of a fully connected (FC) layer and an output layer. All 512 neurons of the FC layer connect to each of the 17 neurons in the output layer. To bound the neuron values of the FC layer, ReLU activation is applied. The output layer produces prediction probabilities, corresponding to the 17 unique labels, using sigmoid activation.

To prevent overfitting, dropout regularization is applied to each CCM super-layer and the FC layer. More specifically, the CMM super-layers have a dropout rate of 0.25, while the fully connected layer has a dropout rate of 0.5.

Additionally, the input layer and the FC layer are normalized using Batch Normalization [15]. This makes the distribution of each batch identical with a mean of \(\mu = 0\) and a standard deviation of \(\sigma = 1\), allowing for the use of higher learning rates.

Lastly, the binary cross-entropy loss function is used to measure the error during training. This loss function was minimized using the Adam optimization algorithm [16] with continuously decreasing learning rates as the training progresses and the learning ability of the architecture plateaus. Details regarding the training procedure are presented in Section 2.3.

Refer to Appendix A.4 for a detailed Python algorithm based on the custom CNN architecture.
Figure 2.2: Custom CNN Architecture
2.2.2 Pre-trained CNN Architectures

To supplement the custom CNN architecture, several other well-known pre-trained CNN architectures, such as DenseNet [17], VGG [18], ResNet [19], Xception [20], and Inception [21], were implemented. These pre-trained architectures were modified such that they could be applied to the task at hand. More specifically, as illustrated in Figure 2.3, their classification block was removed and replaced by the classification block of the custom CNN architecture discussed on the previous page.

![Figure 2.3: Pre-trained CNN Architecture Design](image)

Unlike the custom CNN architecture, which was trained from scratch, these pre-trained CNN architectures were initialized with the weights produced by corresponding architectures in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [22]. Once trained on the training data-set, these pre-trained CNN architectures along with the custom CNN architecture generated prediction labels for the test data-set, achieving the $F_2$ scores listed in Table 2.3.

Table 2.3: $F_2$ Scores of Various Architectures

<table>
<thead>
<tr>
<th>Architecture</th>
<th>$F_2$ score (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet50</td>
<td>92.886</td>
</tr>
<tr>
<td>VGG16</td>
<td>92.785</td>
</tr>
<tr>
<td>VGG19</td>
<td>92.747</td>
</tr>
<tr>
<td>DenseNet121</td>
<td>92.717</td>
</tr>
<tr>
<td>Custom</td>
<td>92.677</td>
</tr>
<tr>
<td>DenseNet161</td>
<td>92.446</td>
</tr>
<tr>
<td>Xception</td>
<td>92.422</td>
</tr>
<tr>
<td>ResNet34</td>
<td>92.258</td>
</tr>
<tr>
<td>ResNet101</td>
<td>92.132</td>
</tr>
<tr>
<td>InceptionV3</td>
<td>91.888</td>
</tr>
</tbody>
</table>

Refer to Appendix A.5 for design information based on the ResNet50 pre-trained CNN architecture
2.3 Training Procedure

From the 40,479 JPEG images provided in the training data-set, 20% (8,096) were randomly selected to be the validation set, while the remaining 80% (32,383) were used as the training set. All the CNN architectures described in Section 2.2 were trained using the Adam optimization algorithm with batch sizes of 16 for deep architectures all the way up to 128 for shallow architectures.

The learning rate was initially set to $5 \cdot 10^{-4}$ and got reduced by a factor of 10 if the loss function of the validation set was not reduced for 2 consecutive iterations (epochs) through the training set. Additionally, lists containing 3 elements were made for both the learning rate and number of epochs, allowing the architectures to train using the 3 “index-wise” parameter combinations. For each of the 3 learning rate and number of epochs combinations, if the loss function of the validation set was not minimized for 3 consecutive epochs, the next combination cycle began. This was a necessary step which prevented potential overfitting related issues and allowed the architectures to slightly improve their prediction results.

Furthermore, the weights corresponding to the epoch which led to the minimized loss function of the validation set were saved. These saved weights could then be reloaded into the architecture at any time to generate label predictions, without having to re-train the architecture. Lastly, the training was stopped if the loss function of the validation set plateaus for 3 consecutive epochs on the third learning rate and number of epochs combination.

The $F_2$ score and loss function for the training and validation sets of the ResNet50 [19] architecture are shown in Figure 2.4. Note that in Figures 2.4a & 2.4b the red vertical dotted lines show where new learning rate and number of epochs combinations began.

![Figure 2.4: ResNet50 Training Stage Performance Log](image-url)
2.3.1 Data Augmentation

As seen in Section 2.1, there is bias present in the distribution of labels among the training images, which significantly influences the ability of any algorithm to make accurate label predictions on the test images. The effects of this label distribution bias on the training stage of the ResNet50 [19] architecture are shown in Figure 2.5.

![Figure 2.5: Effects of Label Distribution Bias on Training Stage](image)

Figure 2.5 shows that infrequent labels, such as ‘slash_burn’, are predicted for very few of the training images with these predictions being rarely correct. Whereas, frequent labels, such as ‘primary’, are predicted for almost every training image with high precision.

To increase the diversity of the training data-set and improve generalization onto the test data-set, various modifications were applied to the training images in a process called augmentation [23]. For each implemented CNN architecture, the provided training data-set was augmented in a “batch-wise” manner by applying random horizontal and vertical flipping, zooming from 0.8 to 1.2 times the original size, and rotating up to ±90°.

Refer to Appendix A.6 for a Python script showing the training procedure with data augmentation.
It is important to note that for rotated images, any gaps created by the rotation are filled in through reflection, as demonstrated in Figure 2.6.

![Figure 2.6: Reflection for Filling Gaps Formed by Rotation](image)

2.3.2 Optimal Threshold Selection

For every test image, a given CNN architecture generates real values between 0 and 1 for each of the 17 neurons of its output layer. In order to obtain binary label prediction vectors from these “raw” label prediction vectors, thresholds need to be applied. These thresholds were found using the training data-set based raw label predictions, produced by each CNN architecture, and the corresponding ground truth binary label vectors. More specifically, the optimal threshold for a given label index was determined by sweeping its value between 0 and 1 in steps of 0.02, while keeping the thresholds of all other label indices constant, and observing the corresponding $F_2$ score on the training images. If a given threshold value increased the $F_2$ score it was stored, otherwise it was ignored and the threshold value sweep continued. This process allowed only the optimal thresholds corresponding to each label index to be stored in the final 17-dimensional threshold vector.

Refer to Appendix A.2.2 for a Python script outlining the threshold optimization procedure
For the ResNet50 [19] architecture the optimal threshold values for each label are shown in Table 2.4.

<table>
<thead>
<tr>
<th>Label</th>
<th>Threshold Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>blow_down</td>
<td>0.16</td>
</tr>
<tr>
<td>bare_ground</td>
<td>0.20</td>
</tr>
<tr>
<td>conventional_mine</td>
<td>0.26</td>
</tr>
<tr>
<td>blooming</td>
<td>0.24</td>
</tr>
<tr>
<td>cultivation</td>
<td>0.22</td>
</tr>
<tr>
<td>artisinal_mine</td>
<td>0.20</td>
</tr>
<tr>
<td>haze</td>
<td>0.18</td>
</tr>
<tr>
<td>primary</td>
<td>0.28</td>
</tr>
<tr>
<td>slash_burn</td>
<td>0.22</td>
</tr>
<tr>
<td>habitation</td>
<td>0.20</td>
</tr>
<tr>
<td>clear</td>
<td>0.20</td>
</tr>
<tr>
<td>road</td>
<td>0.22</td>
</tr>
<tr>
<td>selective_logging</td>
<td>0.22</td>
</tr>
<tr>
<td>partly_cloudy</td>
<td>0.22</td>
</tr>
<tr>
<td>agriculture</td>
<td>0.22</td>
</tr>
<tr>
<td>water</td>
<td>0.22</td>
</tr>
<tr>
<td>cloudy</td>
<td>0.06</td>
</tr>
</tbody>
</table>

2.4 Post Processing

Once the training stage was complete, each CNN architecture generated label predictions for the test data-set. Then a combination of these label predictions was formed using ensemble techniques [2]. It became apparent that label predictions from uncorrelated architectures produced the best ensemble results. In addition, an ensemble containing similar architectures trained on varying input image dimensions marginally improved the label predictions results. This is due to the fact that larger images allow for the detection of fine features, whereas smaller images predict coarse features more accurately. The following presents the two ensemble methods used during this thesis research, namely Weighted Majority Voting and Weighted Soft Voting.

2.4.1 Weighted Majority Voting Ensemble

The Weighted Majority Voting ensemble is shown in Figure 2.7 on the following pages. Every architecture in this ensemble was trained on 256 × 256 JPEG images, other than Xception [20] which was trained on four different resolutions: 96 × 96, 128 × 128, 150 × 150, and 256 × 256. In the figure, Xception_ N indicates that the architecture was fed N × N
input images during the training stage. Additionally, the weights corresponding to each architecture are provided.

For this ensemble method, although the weights must be integer values, there is no specific metric for weight assignment to each architecture. That being said, ResNet50 [19] was given a higher weight than other architectures as it achieved the highest $F_2$ score (Table 2.3).

To provide better understanding, the key steps of this ensemble method are outlined below.

1. Let $y^j$ and $w^j$ be the binary label prediction vector for a given input image and the weight of the $j$-th architecture in Figure 2.7, respectively.

   Form the vector of label votes, $v = (v_0, v_1, ..., v_{16})$, as
   \[ v = \sum_{j=1}^{8} w^j \cdot y^j, \]  
   (2.3)
   where the $j$-th architecture contributes up to $w^j$ votes to the total number of votes for each label.

2. Compute the maximum number of votes, $M$, from the weights in Figure 2.7 using
   \[ M = \sum_{j=1}^{8} w^j = 17. \]  
   (2.4)

3. Create the final label predictions, $y = (y_0, y_1, ..., y_{16})$, by applying a threshold of $M/2 = 8.5$ to each label index. Specifically,
   \[ y_i = \begin{cases} 
   1, & \text{if } v_i > M/2 \\
   0, & \text{else} 
   \end{cases} \]  
   (2.5)
   for label indices $i = 0, 1, ..., 16$.

   It is important to note that although other weight assignments than the ones presented in Figure 2.7 are possible, care must be taken to make $M$ an odd integer. This ensures that weighted votes $v_i$ cannot equal the threshold value of $M/2$.

Refer to Appendix A.8.1 for an algorithm outlining the Weighted Majority Voting Ensemble method.
2.4.2 Weighted Soft Voting Ensemble

The Weighted Soft Voting ensemble is shown in Figure 2.8 on the next page. Unlike the ensemble method discussed above which performs weighted majority voting on the binary label predictions, this ensemble method performs weighted averaging on the raw label predictions from each CNN architecture, before they are thresholded.

For more accurate results, only the differently structured architectures, mentioned in Figure 2.7, that trained on $256 \times 256$ JPEG images were included in this ensemble. This was required as the Weighted Soft Voting ensemble is much more sensitive to the correlation between label predictions of different architectures than the Weighted Majority Voting ensemble.

Once again, the key steps of this ensemble method are provided below.

1. Let $y^j$ and $w^j$ be the raw label prediction vector for a given input image and the weight of the $j$-th architecture in Figure 2.8, respectively.

   Form the weighted label prediction vector, $\bar{x} = (x_0, x_1, ..., x_{16})$, as

   $$\bar{x} = \sum_{j=1}^{5} w^j \cdot y^j$$  \hspace{1cm} (2.6)
2. Let \( t^j \) be the optimal threshold vector of the \( j \)-th architecture in Figure 2.8.

Form the weighted optimal threshold vector, \( \bar{t} = (t_0, t_1, ..., t_{16}) \), as

\[
\bar{t} = \sum_{j=1}^{5} w^j \cdot t^j
\]  

(2.7)

3. Apply the weighted optimal threshold vector, \( \bar{t} \), to the weighted label prediction vector, \( \bar{x} \), to form the final label predictions, \( y = (y_0, y_1, ..., y_{16}) \). Specifically,

\[
y_i = \begin{cases} 
1, & \text{if } x_i > t_i \\
0, & \text{else}
\end{cases}
\]  

(2.8)

for label indices \( i = 0, 1, ..., 16 \).

Most importantly, a Linear Minimum Mean Square Error (LMMSE) \cite{24} inspired algorithm is utilized in this ensemble method to assign weights to each architecture. As shown in Equation 2.9a, this algorithm consists of non-linear operations, whereas LMMSE attempts to minimize the mean square error (MSE) of a given parameter using only linear operations. This removes any of the ambiguity present in the Weighted Majority Voting ensemble method.

The weight assignment developed here takes into account the \( F_2 \) score achieved by each architecture to produce real valued weights whose sum is always 1, simplifying further calculations. For example, although not specified in Equation 2.7 it is implied that the expression given is also divided by the sum of the weights.

The expression for weights used in this ensemble method is given by

\[
w^j = \frac{e^{z(j)}}{\sum_{i=1}^{5} e^{z(i)}}
\]  

(2.9a)

with

\[z(j) = \left( \left(1 - F_2^j \right)^2 \cdot \text{norm} \right)^{-1}\]  

(2.9b)

where \( F_2^j \) is the \( F_2 \) score (Table 2.3) and \( w^j \) is the weight of the \( j \)-th architecture, respectively.

In Equation 2.9a, exponential terms are used to emphasize the performance variation of different architectures. The variable \( \text{norm} \) in Equation 2.9b is used to normalize the output of the exponential term, which allows for weight adjustment control. The weights assigned to each architecture are shown in Figure 2.8 below and correspond to \( \text{norm} = 38 \) which was found as follows.

1. Set \( \text{norm} = 20 \) and record the \( F_2 \) score obtained from the ensemble.
2. Increment the normalization value by 1 up to \( \text{norm} = 50 \), monitoring the \( F_2 \) score.

3. Once the \( F_2 \) score peaks, record the corresponding normalization value.

![Figure 2.8: Weighted Soft Voting Ensemble](image)

### 2.5 Results

The Weighted Majority Voting ensemble, described in Section 2.4.1, achieved an \( F_2 \) score of 92.990\% on the test data-set. The Weighted Soft Voting ensemble, described in Section 2.4.2, achieved an \( F_2 \) score of 93.070\% on the test data-set. In comparison, current state of the art algorithms achieve an \( F_2 \) score of 93.318\% [11], which is only 0.248\% higher than the result obtained by the Weighted Soft Voting ensemble created during this thesis research.

It is interesting to note that both of these ensemble methods produced an \( F_2 \) score that is higher than the \( F_2 \) score of 92.886\% achieved by the best architecture, ResNet50 [19]. This shows that including lower performing, yet uncorrelated, architectures in the ensemble can improve the overall label prediction results.

Additionally, the results obtained by the Weighted Soft Voting ensemble validate the benefits of assigning weights to each architecture using the proposed weight assignment. Moreover, the impact of the order in which thresholds are applied to the architectures is clearly demonstrated by the notable variation in label prediction results between the two ensemble methods presented in Sections 2.4.1 & 2.4.2.

Refer to Appendix A.8.2 for a Python algorithm outlining the Weighted Soft Voting Ensemble method which includes the LMMSE weight assignment approach
For ease of reference, the test data-set based $F_2$ scores mentioned in this section are summarized in Table 2.5 below.

Table 2.5: $F_2$ Scores of Ensemble Methods

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>$F_2$ score (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet50 Architecture</td>
<td>92.886</td>
</tr>
<tr>
<td>Weighted Majority Voting</td>
<td>92.990</td>
</tr>
<tr>
<td>Weighted Soft Voting</td>
<td>93.070</td>
</tr>
<tr>
<td>State of the Art</td>
<td>93.318</td>
</tr>
</tbody>
</table>

Lastly, for each label in the training data-set, the $F_2$ and Jaccard Index scores\footnote{See Chapter 3 for details regarding the Jaccard Index of Similarity accuracy measure} obtained are summarized in Table 2.6.

Table 2.6: Training Data-set Label $F_2$ vs. Jaccard Index Scores

<table>
<thead>
<tr>
<th>Label</th>
<th>$t_p, f_p, t_n, f_n$</th>
<th>$F_2$ (%)</th>
<th>Jaccard Index (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>primary</td>
<td>35695, 2851, 115, 1818</td>
<td>94.632</td>
<td>88.433</td>
</tr>
<tr>
<td>clear</td>
<td>21233, 8965, 3083, 7198</td>
<td>73.766</td>
<td>56.779</td>
</tr>
<tr>
<td>agriculture</td>
<td>4762, 10726, 17438, 7553</td>
<td>36.773</td>
<td>20.668</td>
</tr>
<tr>
<td>road</td>
<td>1956, 7909, 24499, 6115</td>
<td>23.203</td>
<td>12.240</td>
</tr>
<tr>
<td>water</td>
<td>1605, 7147, 25921, 5806</td>
<td>20.901</td>
<td>11.025</td>
</tr>
<tr>
<td>partly_cloudy</td>
<td>1529, 6978, 26240, 5732</td>
<td>20.359</td>
<td>10.738</td>
</tr>
<tr>
<td>cultivation</td>
<td>781, 5870, 30132, 3696</td>
<td>15.900</td>
<td>7.548</td>
</tr>
<tr>
<td>habitation</td>
<td>445, 4504, 32315, 3215</td>
<td>11.358</td>
<td>5.451</td>
</tr>
<tr>
<td>haze</td>
<td>256, 3424, 34358, 2441</td>
<td>8.847</td>
<td>4.182</td>
</tr>
<tr>
<td>cloudy</td>
<td>147, 2940, 35450, 1942</td>
<td>6.423</td>
<td>2.923</td>
</tr>
<tr>
<td>bare_ground</td>
<td>30, 1160, 38457, 832</td>
<td>3.234</td>
<td>1.484</td>
</tr>
<tr>
<td>selective_logging</td>
<td>2, 356, 39783, 338</td>
<td>0.582</td>
<td>0.287</td>
</tr>
<tr>
<td>artisinal_mine</td>
<td>3, 363, 39777, 33</td>
<td>0.871</td>
<td>0.427</td>
</tr>
<tr>
<td>blooming</td>
<td>1, 178, 39969, 331</td>
<td>0.332</td>
<td>0.196</td>
</tr>
<tr>
<td>slash_burn</td>
<td>0, 79, 40191, 209</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>blow_down</td>
<td>0, 66, 40315, 98</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>conventional_mine</td>
<td>0, 89, 40290, 100</td>
<td>0.000</td>
<td>0.000</td>
</tr>
</tbody>
</table>

As can be seen in Table 2.6, the more frequent labels, such as ‘primary’ and ‘clear’, are predicted with much higher $F_2$ and Jaccard Index scores than the less frequent labels. It is also important to note that the Jaccard Index score is lower than the $F_2$ score for each label, making it seem like a more sensitive accuracy measure. This important observation will be confirmed in Chapter 3.
Chapter 3

Landsat 8 Image Analysis System

In satellite imaging, clouds occlude information in the visible spectrum. Therefore, it is important to identify images containing clouds as these are less valuable for detailed analysis. This chapter focuses on the developing of a deep learning based system capable of outperforming current state of the art algorithms on cloud detection in Landsat 8 imagery.

3.1 Landsat 8 Images

For this thesis research, only 4 out of the 11 channels of the Landsat 8 images\(^2\) were utilized, as outlined in Table 3.1.

<table>
<thead>
<tr>
<th>Channel</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ultra Blue (coastal/aerosol)</td>
</tr>
<tr>
<td>2</td>
<td>Blue</td>
</tr>
<tr>
<td>3</td>
<td>Green</td>
</tr>
<tr>
<td>4</td>
<td>Red</td>
</tr>
<tr>
<td>5</td>
<td>Near Infrared (NIR)</td>
</tr>
<tr>
<td>6</td>
<td>Shortwave Infrared (SWIR) 1</td>
</tr>
<tr>
<td>7</td>
<td>Shortwave Infrared (SWIR) 2</td>
</tr>
<tr>
<td>8</td>
<td>Panchromatic</td>
</tr>
<tr>
<td>9</td>
<td>Cirrus</td>
</tr>
<tr>
<td>10</td>
<td>Thermal Infrared (TIRS) 1</td>
</tr>
<tr>
<td>11</td>
<td>Thermal Infrared (TIRS) 2</td>
</tr>
</tbody>
</table>

Unlike the RGB channels used in the Amazon image analysis system presented in Chapter 2, the 4 channels chosen from the Landsat 8 images contain 16 bits per pixel. This time, all of the satellite images provided were labelled, thus the training and test data-sets were manually formed.

\(^2\)https://landsat.usgs.gov/landsat-8-cloud-cover-assessment-validation-data
From the 60 images provided, the first 42 were used as the training set and the remaining 18 formed the test set. Sample test data-set based full resolution images are shown in Figure 3.1, and their corresponding names are indicated for ease of reference.

Figure 3.1: Sample Near-infrared Channel Landsat 8 Images

Figure 3.1 shows each image with equal dimensions, however, this is not the case for the training and test data-sets as the size of each unique image is not fixed. Additionally, as seen in the figure above, all the images are slightly rotated which creates dark gaps that need to be considered.

Refer to Appendix B.9 for detailed information regarding each Landsat 8 image channel


3.1.1 Class Labels

As each of the Landsat 8 images is approximately of size $8000 \times 8000$, they were partitioned into smaller $256 \times 256$ image “chips” in order to avoid memory related issues. This produced 41,481 chips for the training set and 17,828 chips for the test set. Each chip from the training set was assigned one of 2 unique labels, namely ‘cloud’ or ‘clear’, making this a binary classification problem [25]. Table 3.2 summarizes the occurrence frequency of each label among the 41,481 chips present in the training set.

<table>
<thead>
<tr>
<th>Index</th>
<th>Label</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>clear</td>
<td>24,805</td>
</tr>
<tr>
<td>1</td>
<td>cloud</td>
<td>16,676</td>
</tr>
</tbody>
</table>

(3.1)

As seen in Tables 3.2a & 3.2b above, there is bias present in the chip labels for both cases. In this thesis work, the decision was made to include any chips created from the gaps.

3.1.2 Accuracy Measure

In Chapter 2, the $F_2$ accuracy score measure was used to determine how well the given architectures predicted labels for each image. This chapter focuses on improving current state of the art cloud detection/segmentation algorithms, therefore we adopt an accuracy measure that is more commonly used in this area.

This is why the Jaccard Index of Similarity [26] was chosen as an accuracy measure for the Landsat 8 image segmentation task. The general formula for this accuracy measure, shown in Equation 3.1, takes into account both the union and intersection of the predicted (P) and true (T) values.

$$J(P,T) = \frac{|P \cap T|}{|P \cup T|} = \frac{|P \cap T|}{|P| + |T| - |P \cap T|}, \quad 0 \leq J(P,T) \leq 1$$  \hspace{1cm} (3.1)

Equation 3.1 can also be expressed in terms of the familiar number of true positives ($t_p$), false positives ($f_p$), and false negatives ($f_n$):

$$J(P,T) = \frac{t_p}{t_p + f_p + f_n}, \quad 0 \leq J(P,T) \leq 1$$  \hspace{1cm} (3.2)

Comparing Equations 2.2 & 3.2, it can be seen that for the $F_2$ score, its misclassifications ($f_p$ & $f_n$) are scaled down with fractional coefficients, making them not as severe as in the Jaccard Index score. Therefore, although both accuracy measures are always real values between 0 and 1, the Jaccard Index of Similarity is much more sensitive to slight differences between the predicted and true values than the $F_2$ score accuracy measure. Due to this, in practice not many segmentation algorithms can achieve a Jaccard Index score of 80% or higher.
3.2 CNN Binary Classifier

As mentioned previously, there are only two possible labels for each of the input image chips, hence a CNN binary classifier was designed in Keras with Tensorflow backend. In the Amazon satellite image analysis system described in Chapter 2, a 17-dimensional binary label vector was created for each input image. In this system however, it is only necessary to convert the labels into binary scalar values.

Letting $t_i$ be the label of the $i$-th input image chip and $y_i$ be the corresponding binary label, from a total of $n$ training image chips:

$$y_i = \begin{cases} 
1, & \text{if } t_i = \text{cloud} \\
0, & \text{if } t_i = \text{clear}
\end{cases} \quad (3.3)$$

for input image indices $i = 0, 1, ..., n - 1$.

To allow for training on the 4 channel input chips, a custom architecture needed to be designed as the pre-trained Keras architectures only accept 3 channel inputs. Therefore, the structure of the designed CNN binary classifier architecture was chosen to be identical to the custom CNN architecture described in Section 2.2. The minor difference between the two architectures comes from the fact that as this is a binary classifier, only 1 output neuron is needed at the output layer of Figure 2.2.

3.3 Training Procedure

The Landsat 8 image analysis system follows the same training procedure as outlined in Section 2.3. The key steps of this training procedure are provided below.

1. Randomly split the 41,481 image chips, generated from the 42 full resolution training images, into a validation set (20%) and a training set (80%).

2. Apply the Adam optimization algorithm with a batch size of 32 as the custom CNN architecture is relatively deep in nature.

3. Set the learning rate to $5 \cdot 10^{-4}$ and reduce by a factor of 10 once the validation Jaccard Index measure plateaus for 2 consecutive epochs.

4. Set a large value for the number of epochs to allow sufficient training time.

5. Begin the next learning rate and number of epochs combination when the validation Jaccard Index measure does not improve for 3 consecutive epochs.

Refer to Appendices B.2, B.5, & B.6 for a Jaccard Index score, CNN binary classifier, and training procedure scripts, respectively.
6. Record the weights leading to the highest validation set based Jaccard Index score.

7. Stop the training if the validation Jaccard Index measure plateus for 3 consecutive epochs on the final learning rate and number of epochs combination.

Note that in this training procedure, the Jaccard Index was monitored as it is the most relevant measure that can be used to gauge the performance of the designed system. For the same reasons, the loss function utilized during training was set equal to 1−Jaccard Index.

The Jaccard Index and loss function for the training and validation sets of the custom CNN architecture, designed for the Landsat 8 image segmentation task, are shown in Figure 3.2. Once again the red vertical dotted lines, shown in Figures 3.2a & 3.2b, indicate the beginning of a new learning rate and number of epochs combination.

![Figure 3.2: Landsat 8 Custom CNN Training Stage Performance Log](image)

### 3.3.1 Data Augmentation

The basic data augmentation techniques mentioned in Section 2.3.1 were utilized once again in the Landsat 8 image analysis system. More specifically, each batch of chip images randomly received the following image transformations.

1. Flipping in both the horizontal and vertical direction
2. Zooming from 0.8 to 1.2 times the original size
3. Rotating up ±90°
4. Using reflection to fill any gaps formed by rotation
3.3.2 Full Resolution Image Based Threshold Selection

To determine the optimal threshold value for each full resolution image, its corresponding chips were used. Unlike in Section 2.3.2, to successfully operate, this procedure only requires the raw label predictions produced by the custom CNN architecture. Also, as the original full resolution images belong to the test set, this procedure utilizes raw label predictions for test set based chip images. The key steps that were applied to select the optimal thresholds for each full resolution image are provided below.

1. For all the chip images produced from the full resolution input image, find the average value of their raw label predictions. This average value becomes the threshold.

2. Compute the Jaccard Index score when the threshold value is applied.

3. Store the threshold value in a list whose indices correspond to the full resolution input image.

4. Load in the next full resolution image and repeat the whole process.

It is important to note that as there are 18 full resolution images in the test set, the optimal threshold selection procedure outlined above produced 18 unique threshold values.

3.4 Current Results

During the training stage, the training set based validation Jaccard Index measure reached a maximum value of 87.72%, as shown in Figure 3.2a. To illustrate the generalization abilities of the CNN binary classifier on the test data-set, its segmentation performance is shown in Table 3.3.

Table 3.3: CNN Binary Classifier Segmentation Performance

<table>
<thead>
<tr>
<th>Image Label</th>
<th>cloud</th>
<th>clear</th>
<th>cloud</th>
<th>clear</th>
<th>cloud</th>
<th>clear</th>
<th>cloud</th>
<th>clear</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Image</td>
<td>3.1a</td>
<td>3.1b</td>
<td>3.1c</td>
<td>3.1d</td>
<td>3.1a</td>
<td>3.1b</td>
<td>3.1c</td>
<td>3.1d</td>
</tr>
<tr>
<td>Precision (%)</td>
<td>97.732</td>
<td>95.251</td>
<td>99.212</td>
<td>95.455</td>
<td>94.816</td>
<td>97.064</td>
<td>96.458</td>
<td>95.819</td>
</tr>
<tr>
<td>Recall (%)</td>
<td>95.564</td>
<td>95.251</td>
<td>96.762</td>
<td>94.326</td>
<td>97.339</td>
<td>97.063</td>
<td>99.143</td>
<td>96.661</td>
</tr>
</tbody>
</table>

As can be seen in Table 3.3, due to the fact that the ‘clear’ label is present in more chips than the ‘cloud’ label, the binary classifier predicts it more often.

3.4.1 Cloud Mask Generation

Once the label predictions were generated for each chip image in the test data-set, binary valued cloud masks representing the full resolution images were created as follows.

Refer to Appendix B.3 for a threshold selection script
1. For each chip image \((i)\), obtain the coordinates \((x_{i-1}, y_{i-1}, x_i, y_i)\) corresponding to its location in the original image. Here \((x_{i-1}, y_{i-1})\) and \((x_i, y_i)\) represent the upper left and lower right corners of the chip, respectively.

2. In the cloud mask image, use the label prediction to fill in the location indicated by the coordinates obtained in the above step, with the appropriate binary value color \((C_b)\). More specifically, letting \(t_i\) represent the \(i\)-th chip image label

\[ C_b = \begin{cases} 
\text{white, if } t_i = \text{cloud} \\
\text{black, if } t_i = \text{clear} 
\end{cases} \tag{3.4} \]

3. Repeat this process until each patch in the entire cloud mask image is filled with the appropriate binary value color.

To illustrate the effectiveness of the designed CNN binary classifier, its predicted cloud masks are shown next to the ground truth masks in Table 3.4.

Table 3.4: Binary Classifier Cloud Masks
(a) Cloud Masks for Figure 3.1a

<table>
<thead>
<tr>
<th>Type</th>
<th>Ground Truth</th>
<th>Binary Predictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cloud Mask</td>
<td>![Ground Truth Image]</td>
<td>![Binary Predictions Image]</td>
</tr>
</tbody>
</table>

(b) Cloud Masks for Figure 3.1b

<table>
<thead>
<tr>
<th>Type</th>
<th>Ground Truth</th>
<th>Binary Predictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cloud Mask</td>
<td>![Ground Truth Image]</td>
<td>![Binary Predictions Image]</td>
</tr>
</tbody>
</table>
In Table 3.4, each patch in the ground truth cloud mask was assigned a binary value color based on the number of cloud pixels it contains. More specifically, for each $256 \times 256$ patch its corresponding binary value color is determined by

$$C_b = \begin{cases} 
\text{white, if } N_{\text{cloud}} > N^2 \cdot 0.05 \\
\text{black, otherwise} 
\end{cases}$$  \hspace{1cm} (3.5)$$

where $N_{\text{cloud}}$ and $N^2$ represent the number of cloud pixels and the total number of pixels in the given patch, respectively.

Additionally, in Table 3.4 there are a few misclassified patches present in the binary prediction masks. On the following page, Table 3.5 shows the raw label predictions based cloud masks to provide insight into how severe the misclassification cases are. These cloud masks illustrate which patches were misclassified as a result of their raw prediction value being on the wrong side of the threshold value.

Refer to Appendix B.8 for a cloud mask generation script
For each cloud mask presented, the corresponding confusion matrix is provided. The confusion matrix, shown in Figure 3.3, is of size $2 \times 2$ and displays all of the information needed to compute the Jaccard Index score.

Table 3.5: Raw Cloud Masks & Confusion Matrix

(a) Raw Cloud Mask & Confusion Matrix for Figure 3.1a

<table>
<thead>
<tr>
<th>Type</th>
<th>Raw Predictions</th>
<th>Confusion Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cloud Mask</td>
<td><img src="Image" alt="Raw Cloud Mask" /></td>
<td><img src="Image" alt="Confusion Matrix" /></td>
</tr>
</tbody>
</table>

(b) Raw Cloud Mask & Confusion Matrix for Figure 3.1b

<table>
<thead>
<tr>
<th>Type</th>
<th>Raw Predictions</th>
<th>Confusion Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cloud Mask</td>
<td><img src="Image" alt="Raw Cloud Mask" /></td>
<td><img src="Image" alt="Confusion Matrix" /></td>
</tr>
</tbody>
</table>
As mentioned above, the Jaccard Index score for each of the generated binary cloud mask can be calculated from Equation 3.2. All of the relevant information needed for the Jaccard Index score calculations can be found in the confusion matrices provided in Table 3.5. The Jaccard Index scores of the cloud masks generated for the original full resolution are summarized in Table 3.6.

<table>
<thead>
<tr>
<th>Cloud Mask for Image</th>
<th>Jaccard Index Score (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 3.1a</td>
<td>93.490</td>
</tr>
<tr>
<td>Figure 3.1b</td>
<td>90.932</td>
</tr>
<tr>
<td>Figure 3.1c</td>
<td>96.030</td>
</tr>
<tr>
<td>Figure 3.1d</td>
<td>90.271</td>
</tr>
</tbody>
</table>
As can be seen in Table 3.6 above, the Jaccard Index scores for all 4 cloud mask images (shown in Table 3.4) are significantly higher than the validation Jaccard Index score of 87.72% obtained during the training stage. Thus, when provided with input 256 × 256 chip images, the designed CNN binary classifier generalizes very well. More specifically, it is able to generate highly accurate 256 × 256 patch based cloud masks for full resolution test images which it never saw previously.

Refer to Appendices B.8.1 & B.8.2 for raw prediction cloud mask and confusion matrix generation scripts, respectively
Chapter 4

Conclusion & Future Work

This thesis research focused on designing two deep learning based systems for satellite image analysis. The first system focuses on providing multi-label classification for satellite images of the Amazon. The second system analyzes Landsat 8 imagery to detect any clouds present in the images. Both systems aim to accurately distinguish between various features of satellite images in an efficient manner.

In Chapter 2, the Amazon image analysis system was designed. This system utilized a combination of custom and pre-trained CNN architectures. In this task each of the 3 channel 256 × 256 input JPEG images had 17 unique labels. The best architecture, ResNet50 [19], achieved an $F_2$ score of 92.886%. Once all architectures were trained, two ensemble techniques were introduced, namely Weighted Majority Voting and Weighted Soft Voting.

In the Weighted Majority Voting ensemble, integer valued weights were assigned based on the performance of each architecture, with the sum of the weights being an odd value. These weights were applied to the binary labels produced for each class. On the contrary, in the Weighted Soft Voting ensemble the weights were assigned to better correspond with individual model’s performance. This produced real valued weights whose sum is always 1. In addition, this removed any of the ambiguity present in the weight selection of the Weighted Majority Voting ensemble. These weights were applied to raw prediction scores before thresholding.

It was found that including high performance, yet uncorrelated, architectures in the ensemble significantly increases the overall results. The Weighted Soft Voting ensemble method produced the best results, achieving an $F_2$ score of 93.070%. This result is only 0.248% lower than current state of the art algorithms.

Any further developments for this system should focus on improving each individual CNN architecture used in the ensemble. This is feasible as current state of art algorithms are capable of achieving an $F_2$ score greater than 93% with a single architecture. Additionally, the implementation of other common ensemble techniques should be considered, as they can lead to even better results. Moreover, for the Weighted Soft Voting ensemble method
presented in Section 2.4.2, its weighted optimal threshold vector can be re-evaluated using the procedure outlined for each individual architecture (Section 2.3.2). Furthermore, rather than applying a single weight to all the labels of an architecture, a weight matrix can be formed to apply label-specific weights for each architecture. Lastly, an accuracy measure that is more sensitive to a specific label can be applied. This will allow the system to generalize to tasks which are only interested in a subset of the 17 labels mentioned in Table 2.2a.

In Chapter 3, the Landsat 8 image analysis system was introduced. Here the input images varied in size and were approximately $8000 \times 8000$, therefore $256 \times 256$ chips were created from each of the full resolution images. Unlike the Amazon image analysis system, which trained on 3-channel input JPEG images, this system utilized a custom CNN binary classifier to train on the 4-channel input chips. From the test data-set based predictions generated by the binary classifier for each $256 \times 256$ chip, full resolution cloud masks were created and compared to the ground truth cloud masks. Then the accuracy of the predicted cloud masks was measured using the Jaccard Index of Similarity.

The results obtained suggest that the CNN binary classifier outperforms current state of the art segmentation algorithms. However, it is important to remember that these results are for cloud masks formed by predictions made on $256 \times 256$ chip images.

The next step for this research would be to design an architecture capable of generating label predictions for individual pixels. Additionally, new training and test data-sets should be incorporated to include more recent and better quality full resolution Landsat 8 images. Moreover, other sophisticated data augmentation techniques, such as Gaussian Noise [27], Poisson Noise [28], and Salt-and-pepper Noise [29], should be introduced to manipulate the pixel values of the input images. Lastly, for both systems it is important to check what effect each data augmentation technique has on the respective accuracy measure. This will provide information regarding which augmentation techniques need to be avoided and/or included for the optimal system performance.

Overall, the designed image analysis systems show very promising results when compared to current state of the art algorithms. Further improvements to the architectures used in these systems will lead to superhuman performance on related segmentation applications. Any superior deep learning based systems arising from this research can provide crucial insight into the deforestation process, which will help better understand and potentially prevent its negative impact on the environment and ecosystem.
References


Appendix A

Amazon Image Analysis System

To successfully run the algorithm presented in this Appendix, copy it to a Python script named “amazon_cnn_architecture.py”. Then load the required training and test data-sets from [11] into the appropriate directories.

To execute the training stage, use the command:
> python amazon_cnn_architecture.py -mode train.

Likewise, to execute the test stage, use the command:
> python amazon_cnn_architecture.py -mode test.

A.1 Load Required Modules

```python
import numpy as np # Linear algebra
import pandas as pd # Data processing, CSV file I/O (e.g. pd.read_csv)
import cv2 # OpenCV for image manipulation
import os # To get path to current directory

# Define model related parameters
from keras import optimizers
from keras.models import Sequential, Model
from keras.layers import Input, Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D, BatchNormalization
from keras.callbacks import EarlyStopping, ModelCheckpoint, ReduceLROnPlateau
from keras.preprocessing.image import ImageDataGenerator # Data augmentation
from sklearn import cross_validation # Create training/validation set
from sklearn.metrics import fbeta_score # F2 score accuracy measure

# Parsing arguments for Network definition (train vs test)
import argparse
ap = argparse.ArgumentParser()
ap.add_argument('-mode', default='train')
args = vars(ap.parse_args())
mode = args['mode']
```

40
A.2  Function Definitions

A.2.1  Accuracy Measure

```python
# F2 accuracy score metric
import tensorflow as tf

def f2_score(y_true, y_pred):
    y_true = tf.cast(y_true, "float32")
    cond = tf.greater(y_pred, tf.scalar_mul(0.2, tf.ones(tf.shape(y_pred))))
    y_pred = tf.where(cond,
                       tf.ones(tf.shape(y_pred)),
                       tf.zeros(tf.shape(y_pred))
    )
    y_pred = tf.cast(tf.round(y_pred), "float32")
    y_correct = y_true * y_pred
    sum_true = tf.reduce_sum(y_true, axis=1)
    sum_pred = tf.reduce_sum(y_pred, axis=1)
    sum_correct = tf.reduce_sum(y_correct, axis=1)
    precision = sum_correct / sum_pred
    recall = sum_correct / sum_true
    f_score = 5 * precision * recall / (4 * precision + recall)
    f_score = tf.where(tf.is_nan(f_score), tf.zeros_like(f_score), f_score)
    return tf.reduce_mean(f_score)
```

A.2.2  Threshold Optimization

```python
# Threshold Optimization Function

def get_optimal_threshold(true_label, prediction, iterations = 50):  
    best_threshold = [0.2]*17  # Define initial thresholds (0.2 for each label)
    for t in range(17):
        best_fbeta = 0
        temp_threshold = np.array(best_threshold)

        # For each label index:
        # 1. Go through threshold values 0 to 1 in steps of 1/50
        # 2. Compute the F2 score with that threshold
        # 3. If F2 score > previous, save new threshold value.
        for i in range(iterations):
            temp_value = i / float(iterations)
            temp_threshold[t] = temp_value
            temp_fbeta = fbeta_score(true_label, prediction >= temp_threshold,
                                      beta=2, average='samples')
            if(temp_fbeta >= best_fbeta):
                best_fbeta = temp_fbeta
                best_threshold[t] = temp_value
        return best_threshold
```
A.3 Image & Label Preparation

```python
# Define train and test input parameters
x_train = []
x_test = []
y_train = []

# Read in training and test CSV files
df_train = pd.read_csv('../train_v2.csv')
df_test = pd.read_csv('../sample_submission_v2.csv')

# Flatten the 'tags' column of the training data-set into a list
flatten = lambda l: [item for sublist in l for item in sublist]
labels = list(set(flatten([l.split(' ') for l in df_train['tags'].values])))

labels = ['agriculture',
          'artisinal_mine',
          'bare_ground',
          'blooming',
          'blow_down',
          'clear',
          'cloudy',
          'conventional_mine',
          'cultivation',
          'habitation',
          'haze',
          'partlycloudy',
          'primary',
          'road',
          'selective_logging',
          'slash_burn',
          'water']

label_map = {'blow_down': 0,
              'bare_ground': 1,
              'conventional_mine': 2,
              'blooming': 3,
              'cultivation': 4,
              'artisinal_mine': 5,
              'haze': 6,
              'primary': 7,
              'slash_burn': 8,
              'habitation': 9,
              'clear': 10,
              'road': 11,
              'selective_logging': 12,
              'partly_cloudy': 13,
              'agriculture': 14,
              'water': 15,
              'cloudy': 16}
```
A.4 Custom CNN Architecture

```python
# Input shape = (dim, dim, # of channels)
model = Sequential()

# Input Layer
model.add(BatchNormalization(input_shape=(dim, dim, 3)))

# CCM_1
model.add(Conv2D(32, (3, 3), padding='same', activation='relu'))
model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

# CCM_2
model.add(Conv2D(64, (3, 3), padding='same', activation='relu'))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

# CCM_3
model.add(Conv2D(128, (3, 3), padding='same', activation='relu'))
model.add(Conv2D(128, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

# CCM_4
model.add(Conv2D(256, (3, 3), padding='same', activation='relu'))
model.add(Conv2D(256, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

# Create a feature vector from the CCM_4 final layer
model.add(Flatten())

# Fully Connected (FC) Layer
model.add(Dense(512, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))

# Output Layer
model.add(Dense(17, activation='sigmoid'))
```

43
A.5 Pre-trained CNN architectures

The following is provided to outline the procedure required to modify the existing algorithm such that other pre-trained CNN architectures can be used.

Firstly, on top of the modules loaded in the custom CNN architecture (Appendix A.1), the modules corresponding to the pre-trained architecture also need to be loaded:

```python
# Loading the pre-trained ResNet50 architecture module
from keras.applications.resnet50 import ResNet50
```

Then the pre-trained architecture is formed, making sure the appropriate input shape is passed by adjusting the “dim” variable.

```python
# Pre-trained architecture Example: ResNet50

# Extract the pre-trained architecture
# without including the classification stage
base_model = ResNet50(input_shape=(dim, dim, 3),
                      include_top=False,
                      weights='imagenet')

# Get the output of the base_model formed above
x = base_model.output

# Flatten to obtain a feature vector
x = Flatten()(x)

# Connect the feature vector to to the fully connected (FC) layer
x = Dense(512, activation='relu')(x)

# Form the output label predictions
predictions = Dense(17, activation='sigmoid')(x)

# Combine the pre-trained architecture ("base_model")
# with the designed classification stage,
# forming the final pre-trained architecture ("model")
model = Model(inputs=base_model.input, outputs=predictions)
```
A.6 Model Training

```python
if(mode == 'train'):
    # Read in the training data-set images
    for f, tags in tqdm(df_train.values, miniters=1000):
        img = cv2.imread('../train-jpg/{}.jpg'.format(f))
        targets = np.zeros(17)

        # Create the 17-dimensional binary label vectors
        for t in tags.split(' '):
            targets[label_map[t]] = 1

        x_train.append(cv2.resize(img, (dim, dim)))
        y_train.append(targets)

    # Normalize the data-sets
    y_train = np.array(y_train, np.uint8)
    x_train = np.array(x_train, np.float32)/255.

    print(x_train.shape)
    print(y_train.shape)

    # Define the train and validation data
    nfolds = 5
    num_fold = 0
    kf = KFold(len(y_train), n_folds=nfolds, shuffle=True, random_state=1)

    for train_index, test_index in kf:
        # Define data augmentation
        datagen = ImageDataGenerator(horizontal_flip=True,
                                      vertical_flip=True,
                                      zoom_range=0.2,
                                      rotation_range=90,
                                      fill_mode='reflect')

        num_fold += 1
        print('Start KFold number {} from {}'.format(num_fold, nfolds))
        print('Split train: ',
              len(x_train[train_index]),
              len(y_train[train_index]))
        print('Split valid: ',
              len(x_train[test_index]),
              len(y_train[test_index]))

    # Path at which ModelCheckpoint callback will store weights
    kfold_weights_path = os.path.join('', 'kfold_{}.h5'.format(num_fold))

    # Train on the data-set at a specific learning rate
    # for the corresponding number of epochs
    epochs_arr = [250, 200, 150]
    learn_rates = [0.0005, 0.00005, 0.000005]

    for learn_rate, epochs in zip(learn_rates, epochs_arr):
        # Compile the model
        opt = optimizers.Adam(lr=learn_rate)
        model.compile(loss='binary_crossentropy',
                      optimizer=opt,
                      metrics=[f2_score])
```

45
# Define stopping, weight saving, and learning rate reduction criteria
callbacks = [EarlyStopping(monitor='val_loss', patience=3, verbose=0),
             ModelCheckpoint(kfold_weights_path, monitor='val_loss',
                             save_best_only=True, verbose=0),
             ReduceLROnPlateau(monitor='val_loss', factor=0.1,
                               patience=2, verbose=0,
                               mode='auto', epsilon=0.0001,
                               cooldown=0, min_lr=0)]

model.fit_generator(datagen.flow(x_train[train_index],
                                  y_train[train_index],
                                  batch_size=24),
                    steps_per_epoch=len(x_train[train_index])/32,
                    validation_data=datagen.flow(x_train[test_index],
                                                  y_train[test_index],
                                                  batch_size=24),
                    validation_steps=len(x_train[test_index])/32,
                    epochs=epochs,
                    callbacks=callbacks,
                    verbose=1)

# Load the best saved weights
if os.path.isfile(kfold_weights_path):
    model.load_weights(kfold_weights_path)

# Check validation accuracy of the model at the end of each KFold
p_valid = model.predict(x_train[test_index],
                        batch_size = 128,
                        verbose=2)
print(fbeta_score(y_train[test_index],
                  np.array(p_valid) > 0.2,  
                  beta=2,  
                  average='samples'))

# Optimal Threshold Computation
print("[INFO] Determining the optimal thresholds for each tag")
best_threshold = []
p_train = model.predict(x_train, batch_size =128, verbose=2)
best_threshold = get_optimal_threshold(y_train, p_train)
print(best_threshold)
A.7 Model Testing - Label Predictions

```
if (mode == 'test'):
    # Read in the test data-set image ground truth labels
    for f, tags in tqdm(df_train.values, miniters=1000):
        targets = np.zeros(17)

        # Create the 17-dimensional binary label vectors
        for t in tags.split(' '):
            targets[label_map[t]] = 1
        y_train.append(targets)

    # Read in the test data-set images
    for f, tags in tqdm(df_test.values, miniters=1000):
        img = cv2.imread('../test-jpg/{}.jpg'.format(f))
        x_test.append(cv2.resize(img, (dim, dim)))

    # Normalize the images
    y_train = np.array(y_train, np.uint8)
    x_test = np.array(x_test, np.float32) / 255.0 for custom CNN

    print(x_test.shape)
    print(y_train.shape)

    # Holds predictions on test data-set for each KFold iteration
    yfull_test = []

    # Define the train and validation data
    nfolds = 5
    num_fold = 0
    kf = KFold(len(y_train), n_folds=nfolds, shuffle=True, random_state=1)

    for train_index, test_index in kf:
        num_fold += 1
        print('Start KFold number {} from {}'.format(num_fold, nfolds))

        # Load weights and compile the model
        kfold_weights_path = os.path.join('', 'kfold_{}.h5'.format(num_fold))
        if os.path.isfile(kfold_weights_path):
            model.load_weights(kfold_weights_path)

            opt = optimizers.Adam(lr=0.00001)
            model.compile(loss='binary_crossentropy',
                           optimizer=opt,
                           metrics=[f2_score])

        # Make label predictions from the test data-set
        p_test = model.predict(x_test, batch_size=128, verbose=2)
        yfull_test.append(p_test)

        # Average all the KFolds
        result = np.array(yfull_test[0])
        for i in range(1, nfolds):
            result += np.array(yfull_test[i])
        result /= nfolds
        result = pd.DataFrame(result, columns=labels)
```
Next, to create the submission file replace the best_threshold list found below with the list produced at the end of the training stage.

```python
if (mode == 'test'):  # Continued
    # Create Submission File
    best_threshold = [0.2]*17  # Replace with output list of training stage
    preds = []
    for i in tqdm(range(result.shape[0]), miniters=1000):
        a = result.ix[[i]]
        a = a.apply(lambda x: np.array(x) > np.array(best_threshold), axis=1)
        a = a.transpose()
        a = a.loc[a[i] == True]
        preds.append(" ".join(list(a.index)))

    # Record the label predictions and save to file
    df_test[‘tags’] = preds
    df_test.to_csv(‘Results.csv’, index=False)
```
A.8 Ensemble Procedures

To successfully run these ensemble algorithms, every architecture label prediction CSV file needs to be placed in the same folder as the Python script.

A.8.1 Weighted Majority Voting Ensemble

1. Copy the algorithm provided below into a Python Script named “wmv_ensemble.py”.

2. Make sure that each file name follows the format: '_w{weight}_filename.csv'

3. Execute the script using the command:
   
   > python wmv_ensemble.py "_*w\d*.csv" "mwv_ensemble.csv"
   
   The resulting ensemble will thus be stored in a file named “wmv_ensemble.csv”.

```python
from collections import defaultdict, Counter
from glob import glob
import sys, re

# Store first and second command arguments in corresponding variable
files = sys.argv[1]
output = sys.argv[2]

# Extract the integer valued weight from the file name
weight = re.compile(r".*_w\d*.*\d*\.*")
votes = defaultdict(list)  # Used to store the voted labels

with open(output, "wb") as output_file:
    # Create a weight list of 1's with length equal to number of files
    weight_list = [1]*len(glob(files))

    for i, input_file in enumerate(glob(files)):
        print("file: {}, weight: {}\n".format(input_file, weight))
        weight_list[i] = weight_list[i]*int(weight)  # Adjust the weight list

        # Sort input_file by first column and ignore the header
        lines = open(input_file).readlines()
        lines = [lines[0]] + sorted(lines[1:])

        for e, line in enumerate(lines):
            # If line is header then write it directly to output file
            # Else append the line as many times as specified by weight
            if i == 0 and e == 0:
                outfile.write(line)
            if e > 0:
                row = line.strip().split(",")
                for x in range(weight_list[i]):
                    votes[(e, row[0])].append(row[1])

        # Extract the label which was voted for the most
        for j, k in sorted(votes):
            most_voted = Counter(votes[(j,k)]).most_common(1)[0][0]
            output_file.write("\n{}").format(most_voted))

        print("Weighted Majority Voting ensemble output to {}\n".format(output))
```

49
A.8.2 Soft Weighted Voting Ensemble

1. Copy the algorithm provided below into a Python Script named “wsv_ensemble.py”.

2. Execute the script using the command:
   > python wsv_ensemble.py “*.csv” “msv_ensemble.csv”.
   The resulting ensemble will thus be stored in a file named “wsv_ensemble.csv”.

```python
from collections import defaultdict, Counter
from glob import glob
import sys, re
import numpy as np  # linear algebra
import pandas as pd  # data processing, CSV file I/O (e.g. pd.read_csv)
from operator import add
from math import expm1, pow

# Store first and second command arguments in corresponding variable
files = sys.argv[1]
outfile = sys.argv[2]

# Dictionary of architectures (filenames) and their corresponding accuracy
model_map = {'ResNet50.csv': 0.92886,
             'VGG16.csv': 0.92785,
             'VGG19.csv': 0.92747,
             'DenseNet121.csv': 0.92717,
             'Xception.csv': 0.92422}

weight_map = {}
key_list = list(model_map.keys())  # Get architecture names from dictionary
_norm = 38  # Normalization factor to control the spread of final weights

# Update the errors of the model map
for i in range(len(model_map)):
    z_x = pow(1 - model_map[key_list[i]], -2) / _norm  # From Equation 2.9b
    model_map.update({key_list[i]: expm1(z_x + 1)})  # expm1 returns e^x - 1

    denominator = sum(model_map.values())

# Update the weight_map
for j in range(len(model_map)):
    weight_map.update({key_list[j]: model_map[key_list[j]] / denominator})

print(weight_map)

initial_raw_output = []  # Weighted raw output of first file
final = np.zeros((61191, 17))  # Weighted raw output
image_name = []

with open(outfile, "wb") as output:
    # Initialize a weight list with 1's and length equal to number of files
    weight_list = [1] * len(glob(files))

    for i, in_file in enumerate(glob(files)):
        weight = weight_map[in_file]
        weight_list[i] = weight_list[i] * weight  # Adjust the weight list

    print("file: {0}, weight: {1}".format(in_file, weight))
    output.write(in_file.encode() + weight_list[i] + final)  # Write weight list and final output
```

# sort glob_file by first column, ignoring the first line
lines = open(in_file).readlines()
for e, line in enumerate(lines):
    # Header
    if i == 0 and e == 0:
        output.write(line)
    # First file
    if i == 0 and e > 0:
        row = line.strip().split(",")
        # Store the image names to write them to file
        image_name.append(row[0])
        # Store the raw predictions from each raw
        row = row[1].split(" ")
        row = [float(p) for p in row]
        # Multiply each index by the appropriate weight
        initial_raw_output.append([x * weight_list[i] for x in row])
    # Next files
    if i >= 1 and e > 0:
        row = line.strip().split(",")
        row = row[1].split(" ")
        row = [float(p) for p in row]
        # Element wise addition of each row from all files
        if i == 1:
            final[e-1] = map(add,
                initial_raw_output[e-1],
                [x * weight_list[i] for x in row])
        else:
            final[e-1] = map(add,
                final[e-1],
                [x * weight_list[i] for x in row])
    # Average the weighted sum and write results to file
    for k in range(61191):
        final[k] = [x/sum(weight_list) for x in final[k]]
        soft_voted = " ".join(map(str, final[k]))
        output.write("{},{}
".format(image_name[k], soft_voted))
        print("Weighted Soft Voting ensemble output to {}".format(outfile))
Appendix B

Landsat 8 Image Analysis System

To successfully run the algorithm presented in this Appendix, copy it to a Python script named “landsat_cnn_architecture.py”. Then load the required training and test data-sets into the appropriate directories.

These data-sets can be found at: https://landsat.usgs.gov/landsat-8-cloud-cover-assessment-validation-data.

To execute the training stage, use the command:
> python landsat_cnn_architecture.py -mode train.

Likewise, to execute the test stage, use the command:
> python landsat_cnn_architecture.py -mode test.

Note that any other stage, such as cloud mask generation, can be executed by specifying the corresponding mode type in the command line.

B.1 Additional Modules Required

It is important to include the following modules in addition to the original modules mentioned in Appendix A.1.

```python
import tifffile as tiff  # This system deals with "TIF" image files
import pylab as pl       # For plotting functionality
import seaborn as sn     # For real value cloud masks and confusion matrices
import matplotlib.colors as mcolors # To create own color map
```
B.2 Jaccard Index of Similarity

```python
# First Method
def compute_jaccard_index(y_true, y_pred):
    intersection = np.logical_and(y_true, y_pred)
    union = np.logical_or(y_true, y_pred)

    return intersection.sum() / float(union.sum())

# Second Method
# Modified from:
# https://www.kaggle.com/drn01z3/end-to-end-baseline-with-u-net-keras
def jaccard(y_true, y_pred):
    intersection = K.sum(y_true * y_pred, axis=[0, -1, -2])
    sum_ = K.sum(y_true + y_pred, axis=[0, -1, -2])
    jac = (intersection) / (sum_ - intersection)

    return K.mean(jac)

# Define the loss function for this system
def jaccard_loss(y_true, y_pred):
    jac_mean = jaccard(y_true, y_pred)
    return 1 - jac_mean
```

Additionally, other relevant parameters found in Equation 3.2 are computed as follows.

```python
def perf_measure(y_actual, y_predicted):
    TP = 0  # Number of true positives
    FP = 0  # Number of false positives
    TN = 0  # Number of true negatives
    FN = 0  # Number of false negatives

    # For each predicted chip label adjust the corresponding parameter
    for i in range(len(y_predicted)):
        if (y_predicted[i]==1 and y_actual[i]==1):
            TP += 1
        if (y_predicted[i]==1 and y_actual[i]==0):
            FP += 1
        if (y_predicted[i]==0 and y_actual[i]==0):
            TN += 1
        if (y_predicted[i]==0 and y_actual[i]==1):
            FN += 1

    return (TP, FP, TN, FN)
```
B.3 Threshold Selection

```python
# Jaccard Index Score Function
def get_accuracy(true_label, prediction):
    TP, FP, TN, FN = perf_measure(true_label, prediction)
    jaccard_index_score = TP/float(TP+FP+FN)  # Based on Equation 3.2
    print(jaccard_index_score)

num_chips = []
thresholds = []
files = df_test['Image_Title'].unique()

# Count the number of chips in each full resolution image
chip_counter = 0
for image in files:
    while (chip_counter < len(df_test['Image_Title'].values) and
           df_test['Image_Title'].values[counter] == image):
        chip_counter += 1
    num_chips.append(chip_counter)
print(num_chips)

for j in range(len(num_chips)):
    if j == 0:
        # Extract the correct chip image label predictions
        temp = p_test[j: num_chips[j]]
        # Compute the mean of the raw label predictions (threshold value)
        mean = sum(temp)/float(len(temp))
        # Print corresponding Jaccard Index score
        get_accuracy(y_true[j: num_chips[j]], p_test[j: num_chips[j]]>mean)
    else:
        temp = p_test[num_chips[j-1]: num_chips[j]]
        mean = sum(temp)/float(len(temp))
        get_accuracy(y_true[num_chips[j-1]: num_chips[j]],
                     p_test[num_chips[j-1]: num_chips[j]]>mean)

# Store the threshold values for all full resolution images
thresholds.append(mean)
print(thresholds)
```

B.4 Data-set Preparation

```python
# Define train and test input parameters
x_train = []
x_test = []
y_train = []

# Read in training file and 2 versions of the test file
# test.xlsx -> no labels are provided
# test_gt.xlsx -> labels are provided
df_train = pd.read_excel('../training.xlsx')
df_test = pd.read_excel('../test.xlsx')
df_test_gt = pd.read_excel('../test_gt.xlsx')

label_map = {'clear': 0, 'cloud': 1}
```
B.5 CNN Binary Classifier

Although the binary classifier designed for this system is similar to the custom CNN architecture designed for the Amazon image analysis system, there are some differences which are mentioned in the comments below.

```python
# Chips are of size 256x256
dim = 256

model = Sequential()

# Input Layer - Images have 4 channels in this system
model.add(BatchNormalization(input_shape=(dim, dim, 4)))

# CCM_1
model.add(Conv2D(32, (3, 3), padding='same', activation='relu'))
model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

# CCM_2
model.add(Conv2D(64, (3, 3), padding='same', activation='relu'))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

# CCM_3
model.add(Conv2D(128, (3, 3), padding='same', activation='relu'))
model.add(Conv2D(128, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

# CCM_4
model.add(Conv2D(256, (3, 3), padding='same', activation='relu'))
model.add(Conv2D(256, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

# Create a feature vector from the CCM_4 final layer
model.add(Flatten())

# Fully Connected (FC) Layer - Less neurons as only 1 label is predicted
model.add(Dense(128, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))

# Output Layer - Only 1 neuron needed for binary classifiers
model.add(Dense(1, activation='sigmoid'))

# Information about the model and number of parameters trained
print(model.summary())
```
B.6 Training Stage

```python
# Define variables used during the architecture training
batch_size = 32
weights = os.path.join('', 'weights.h5')

if(mode == 'train'):
    # Extract chip image information to form data-set
    for row, temp_val in enumerate(sorted(glob.glob("../training/*_.tif"))):
        title = df_train['Image_Title'].values[row]
        x_tl = df_train['X_Top_Left'].values[row]
        y_tl = df_train['Y_Top_Left'].values[row]
        x_br = df_train['X_Bottom_Right'].values[row]
        y_br = df_train['Y_Bottom_Right'].values[row]

        image = "../training/{}_{}{}{}{}.tif".format(title, x_tl, y_tl, x_br, y_br)
        img = tiff.imread(image)  # Load in the image
        label = df_train['Image_Tag_Cloud'].values[row]  # Extract the label
        target = label_map[label]  # Convert labels to binary

        # Form the training data-set
        x_train.append(img)
        y_train.append(target)

    # Normalize the images
    y_train = np.array(y_train, np.uint8)
    x_train = np.array(x_train, np.float32)

    print(x_train.shape)
    print(y_train.shape)

    # Define the train and validation sets
    x_train, x_valid, y_train, y_valid = train_test_split(x_train, y_train, test_size=0.2, random_state=42)

    # Define data augmentation
    datagen = ImageDataGenerator(horizontal_flip=True, vertical_flip=True, zoom_range=0.2, rotation_range=90, fill_mode='reflect')

    print('Split train: ', len(x_train), len(y_train))
    print('Split valid: ', len(x_valid), len(y_valid))

    # Compile the model and monitor accuracy and Jaccard Index measures
    opt = optimizers.Adam(lr=0.0005)
    model.compile(loss=jaccard_loss, optimizer=opt, metrics=['accuracy', jaccard])

    # Define stopping, weight saving, and learning rate reduction criteria
    callbacks = [EarlyStopping(monitor='val_jaccard', patience=10, verbose=0),
                 # Add other callbacks here
                 ]
```
ModelCheckpoint(weights,
    monitor='val_jaccard',
    save_best_only=True, verbose=0),
ReduceLROnPlateau(monitor='val_jaccard', factor=0.1,
    patience=4, verbose=0,
    mode='auto', epsilon=0.0001,
    cooldown=0, min_lr=0)]

# Train for the specified number of epochs
model.fit_generator(datagen.flow(x_train,
    y_train,
    batch_size=batch_size),
    steps_per_epoch=len(x_train)/batch_size,
    validation_data=datagen.flow(x_valid,
    y_valid,
    batch_size=batch_size),
    validation_steps=len(x_valid)/batch_size,
    epochs=250,
    callbacks=callbacks,
    verbose=1)
B.7 Test Stage

```python
if (mode == 'test '):
    # Extract chip image information to form data-set
    for row , temp_val in enumerate(sorted(glob.glob("../testing/* .tif"))):
        title = df_test['Image_Title'].values[row]
        x_tl = df_test['X_Top_Left' ].values[row]
        y_tl = df_test['Y_Top_Left' ].values[row]
        x_br = df_test['X_Bottom_Right'].values[row]
        y_br = df_test['Y_Bottom_Right'].values[row]

        image = "../testing/{}/{}_{}_{}_{}.tif".format(title, x_tl,
                                                   y_tl, x_br, y_br)

        img = tifffile.imread(image)
        x_test.append(img)
        x_test = np.array(x_test, np.float32)

    # Form the ground truth binary label vector
    y_true = []
    for row in range(len(df_test_gt.values)):
        label = df_test_gt['Image_Tag_Cloud'].values[row]  # Extract label
        target = label_map[label]  # Convert labels to binary
        y_true.append(target)
    y_true = np.array(y_true, np.uint8)

    print(x_test.shape)
    print(y_true.shape)

    # Load the best saved weights
    if os.path.isfile(weights):
        model.load_weights(weights)

    # Compile the model
    model.compile(loss=jaccard_loss,
                  optimizer=optimizers.Adam(1r=0.0005),
                  metrics=['accuracy', jaccard])

    # Make label predictions on the test data-set
    p_test = model.predict(x_test, batch_size=128, verbose=1)

    num_chips = []
    thresholds = []
    files = df_test['Image_Title'].unique()

    # Count the number of chips per full resolution image
    counter = 0
    for image in files:
        while (counter < len(df_test['Image_Title'].values) and
               df_test['Image_Title'].values[counter] == image):
            counter += 1
        num_chips.append(counter)
        print(num_chips)

    # Find optimal thresholds for each full resolution image
    for j in range(len(num_chips)):
        if j == 0:
```
```python
temp = get_optimal_threshold(y_true[j: num_chips[j]],
                            p_test[j: num_chips[j]])
thresholds . append(temp)
else:
    start = num_chips[j-1]
    end = num_chips[j]
    temp = get_optimal_threshold(y_true[start:end],
                                  p_test[start:end])
thresholds . append(temp)

print(thresholds)

# Create Submission File
preds = []
img_counter = 0
threshold_counter = 0
for image in files:
    while(img_counter < len(df_test['Image_Title'].values) and
           df_test['Image_Title'].values[img_counter] == image):
        if(p_test[img_counter] > thresholds[threshold_counter]):
            preds.append('cloud')
        else:
            preds.append('clear')
        img_counter += 1
        threshold_counter += 1

# Record the label predictions and save to file
df_test['Image_Tag_Cloud'] = preds
df_test.to_csv('Results.csv', index=False)
```
B.8 Cloud Mask Generation

The ground truth and binary value cloud masks are generated using the following algorithm. Note that any differences between the generation process of the two cloud masks are mentioned in the comments.

```python
if(mode == 'cloud_mask'):
    print("Ground Truth/ Binary Prediction Cloud Mask Generation")
    patch = 0
    for image in files:
        # Set the plot parameters
        ax = plt.subplot(111)
        ax.invert_yaxis()
        ax.xaxis.set_ticks_position('top')
        ax.tick_params(direction='out')

        # Read in the chip top left corner coordinates
        x = df_test['X_Top_Left'].values
        y = df_test['Y_Top_Left'].values

        # Read in the full resolution image
        path = '../Landsat_8_Images/{}/{}B5.TIF'.format(image, image)
        img = tiff.imread(path)
        dim_y, dim_x = img.shape

        # Adjust plot based on the dimensions of the full resolution image
        plt.xticks(np.arange(0, dim_x+dim, dim), rotation=90)
        plt.yticks(np.arange(0, dim_y+dim, dim))

        # For ground truth change df_test to df_test_gt (all lines below)
        while(patch < len(df_test['Image_Title'].values) and
              df_test['Image_Title'].values[patch] == image):
            # If chip label = cloud, draw a white 256x256 patch
            # with top left corner at the appropriate (x,y) coordinate
            if(label_map[df_test['Image_Tag_Cloud'].values[patch]] == 1):
                rect = plt.Rectangle((x[patch]-1,y[patch]-1),255,255,color='w')
                ax.add_patch(rect)

            # If chip label = clear, draw a black 256x256 patch
            # with top left corner at the appropriate (x,y) coordinate
            if(label_map[df_test['Image_Tag_Cloud'].values[patch]] == 0):
                rect = plt.Rectangle((x[patch]-1,y[patch]-1),255,255,color='k')
                ax.add_patch(rect)

            patch += 1

        print('{}//{}'.format(patch, len(df_test.values)))

    # Save then clear plot for next cloud mask
    plt.savefig('cloud_mask/{}.png'.format(image), bbox_inches='tight')
    plt.clf()
    plt.cla()
```
B.8.1 Raw Prediction Cloud Masks

```python
if mode == 'raw_cloud_mask':
    print("Raw Predictions Cloud Mask Generation")
    for i in range(len(num_chips)):
        # Read in the full resolution image data and get the dimensions
        image = files[i]
        path = '../Landsat_8_Images/{}_{}/B5.TIF'.format(image, image)
        img = tiff.imread(path)
        dim_y, dim_x = img.shape

        # Create a raw value prediction matrix representing the input image
        if i==0:
            predictions = p_test[i:num_chips[i]]
            y = int(round(dim_y/dim))
            x = int(num_chips[i]/y)
            matrix = np.reshape(predictions, (x,y))
            df_matrix = pd.DataFrame(matrix)
        else:
            predictions = p_test[num_chips[i-1]:num_chips[i]]
            y = int(dim_y/dim)
            z = num_chips[i]-num_chips[i-1]
            if z%y != 0:
                y += 1
            x = int(z/y)
            matrix = np.reshape(predictions, (x,y))
            df_matrix = pd.DataFrame(matrix)

        # Create new plot
        ax = plt.subplot(111)

        # Create own colormap
        for k in range(4):
            if k == 0:
                color1 = pl.cm.binary(np.linspace(0.33*k, 0.33*k, 256))
            elif k == 1:
                color2 = pl.cm.binary(np.linspace(0.33*k, 0.33*k, 256))
                colors = np.vstack((color2, color1))
            else:
                color3 = pl.cm.binary(np.linspace(0.33*k, 0.33*k, 256))
                colors = np.vstack((color3, colors))
        cmap = mcolors.LinearSegmentedColormap.from_list('colormap', colors)

        # Plot the raw label prediction cloud mask
        sns.heatmap(df_matrix, ax=ax,
                    xticklabels=False, yticklabels=False,
                    cmap=cmap, vmin=0, vmax=1,
                    cbar_kws={'ticks': np.linspace(0.0, 1.0, num=5))})

        # Save then clear plot for next cloud mask
        pl.savefig('raw_prediction_cloud_masks/{}.png'.format(image))
        pl.clf()
        pl.cla()
```

61
B.8.2 Confusion Matrix

```python
if (mode == 'confusion_matrix'):
    print("Confusion Matrix Generation")

labels = ['cloud', 'clear'] # Tick labels for confusion matrix

for i in range(len(num_chips)):
    # Create confusion matrix for each full resolution image
    if i == 0:
        end = num_chips[i]
        TP, FP, TN, FN = perf_measure(y_true[i:end], p_test[i:end] > thresholds[i])
        conf_matrix = [[TP, FN], [FP, TN]]
        df_cm = pd.DataFrame(conf_matrix, labels, labels)
    else:
        start = num_chips[i-1]
        end = num_chips[i]
        TP, FP, TN, FN = perf_measure(y_true[start:end], p_test[start:end] > thresholds[i])
        conf_matrix = [[TP, FN], [FP, TN]]
        df_cm = pd.DataFrame(conf_matrix, labels, labels)

    # Create new plot
    ax = pl.subplot(111)
    sn.set(font_scale=1.4) # Label size

    For confusion matrix template:
    annot = np.asarray([['True Positive', 'False Negative'], ['False Positive', 'True Negative']])
    sn.heatmap(df_cm, annot=annot, fmt='', ax=ax, cbar=False)

    # For regular confusion matrix:
    sn.heatmap(df_cm, annot=True, fmt='g', ax=ax)

    # Adjust plot parameters
    ax.xaxis.set_ticks_position('top')
    ax.xaxis.set_label_position('top')
    ax.tick_params(direction='out', labelsize=16, top='off', left='off')
    pl.xlabel('Predicted', fontsize=18)
    pl.ylabel('True', fontsize=18)
    pl.tight_layout(pad=1.08)

    # Save then clear plot for next cloud mask
    pl.savefig('confusion/{}.png'.format(i))
    pl.clf()
    pl.cla()
```

62
B.9 Channel Description

The following table provides more details regarding each of the 11 channels present in Landsat 8 imagery.

<table>
<thead>
<tr>
<th>Band</th>
<th>Name</th>
<th>Wavelength (µm)</th>
<th>Resolution (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Band 1</td>
<td>Ultra Blue (coastal/aerosol)</td>
<td>0.435 - 0.451</td>
<td>30</td>
</tr>
<tr>
<td>Band 2</td>
<td>Blue</td>
<td>0.452 - 0.512</td>
<td>30</td>
</tr>
<tr>
<td>Band 3</td>
<td>Green</td>
<td>0.533 - 0.590</td>
<td>30</td>
</tr>
<tr>
<td>Band 4</td>
<td>Red</td>
<td>0.636 - 0.673</td>
<td>30</td>
</tr>
<tr>
<td>Band 5</td>
<td>Near Infrared (NIR)</td>
<td>0.851 - 0.879</td>
<td>30</td>
</tr>
<tr>
<td>Band 6</td>
<td>Shortwave Infrared (SWIR) 1</td>
<td>1.566 - 1.651</td>
<td>30</td>
</tr>
<tr>
<td>Band 7</td>
<td>Shortwave Infrared (SWIR) 2</td>
<td>2.107 - 2.294</td>
<td>30</td>
</tr>
<tr>
<td>Band 8</td>
<td>Panchromatic</td>
<td>0.503 - 0.676</td>
<td>15</td>
</tr>
<tr>
<td>Band 9</td>
<td>Cirrus</td>
<td>1.363 - 1.384</td>
<td>30</td>
</tr>
<tr>
<td>Band 10</td>
<td>Thermal Infrared (TIRS) 1</td>
<td>10.60 - 11.19</td>
<td>100 × (30)</td>
</tr>
<tr>
<td>Band 11</td>
<td>Thermal Infrared (TIRS) 2</td>
<td>11.50 - 12.51</td>
<td>100 × (30)</td>
</tr>
</tbody>
</table>

The above table along with other relevant information can be found at: https://landsat.usgs.gov/what-are-band-designations-landsat-satellites

B.9.1 Naming Conventions

Each Landsat 8 image has a unique name which follows the format shown in the figure below.

![Figure B.1: Landsat 8 Naming Conventions](image)

More information can be found at: https://landsat.usgs.gov/what-are-naming-conventions-landsat-scene-identifiers