A REPORT

ON

Determining Cloud Cover in Okta units using IR-VIS images

BY

Name of the Student:	**ID No. :**
Harshit Sharma	2018A3PS0839H
Nishant Kumar	2018B1A30653P
Siddhant Barua	2018B5A70919P
Yathaarth Rajiv Jamwal	2018B4A70862P

AT

Indian Meteorological Department, Pune

A Practice School - 1 Station of

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI (June, 2020)

A REPORT

ON

Determining Cloud Cover in Okta units using IR-VIS images

BY

Name of the Student:	ID No. :	Discipline
Harshit Sharma	2018A3PS0839H	B.E. in EEE
Nishant Kumar	2018B1A30653P	MSc. Biology + B.E EEE
Siddhant Barua	2018B5A70919P	MSc. Physics + B.E CSE
Yathaarth Rajiv Jamwal	2018B4A70862P	Msc. Maths + B.E CSE

Prepared in fulfilment of the Practice School – I Course AT
Indian Meteorological Department, Pune

A Practice School I station of

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI (June, 2020)

ACKNOWLEDGMENTS

We would like to use this opportunity to express our gratitude to everyone who supported us throughout the course of the project. We are thankful for their guidance, invaluably constructive criticism and friendly advice during the project work. We are sincerely grateful to them for sharing their truthful and illuminating views on a number of issues related to the project.

We are also grateful to the Indian Meteorological Department for letting us work on this project and providing us with the needed guidance to complete this project successfully. Our progress in this project would not have been possible without their constant guidance and support.

We would especially like to thank our PS Faculty Dr. Rejesh N.A. and our IMD mentor K.N Mohan for their support and guidance in completion of this project.

ABSTRACT SHEET

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE

PILANI (RAJASTHAN)

Practice School Division

Station: Indian Meteorological Department **Centre:** Pune

Duration: 6 Weeks **Date of Start:** 19nd May, 2020

Date of Submission: 25th June, 2020

Title of the Project: Determining Cloud Cover in Okta units using IR-VIS images

ID No./Name/Discipline of the Students:

2018A3PS0839H
Harshit Sharma
B.E. EEE

2018B5A70919P
Siddhant Barua
MSc. Phy + B.E. CSE

2018B1A30653P
Nishant Kumar
MSc. Bio + B.E EEE

2018B4A70862P
Yathaarth Rajiv Jamwal
Msc. Maths + B.E CSE

Name & Designations of the Experts:

Mr. KN Mohan
Scientist 'F'
IMD - Pune

Name of the PS Faculty: Dr. Rejesh N.A.

Key Words: Image Processing, Deep Learning, Convolutional Neural Network, Thresholding, Cloud Cover, Okta

Project Areas: Image Processing and Deep Learning

Abstract :

The objective of the project is to determine the cloud cover of an area and report it in Okta measurement units.
Okta is a standard unit of cloud cover measurement, where 0 Okta indicates clear skies and 8 Oktas indicates fully cloudy sky. In this project, we discuss different implementations to find the cloud coverage in any given image. These are: Using simple thresholding techniques on VIS image, Using IR satellite data from IMD's website, training a Convolutional Neural Network to perform semantic image segmentation and using the UNET neural network architecture for semantic segmentation.
After the completion of this project, we will be able to find the real time cloud cover at an area which would provide helpful data for meteorological predictions.

Signature of Student

Date: 24/06/2020

Signature of PS Faculty

Date: 25/06/2020

CONTENTS

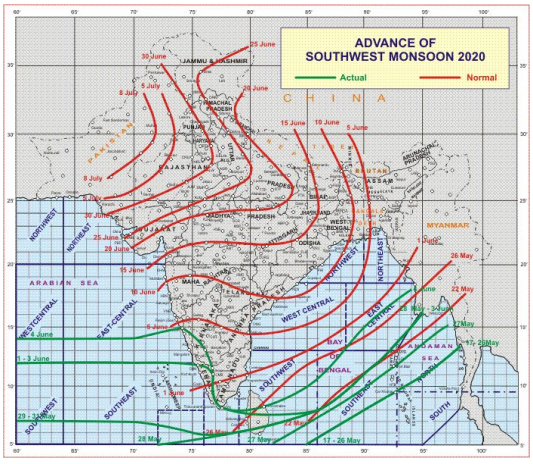
INTRODUCTION

India Meteorological Department (IMD) is an agency of the Ministry of Earth Sciences of the Government of India, formed in 1875. It is the principal agency responsible for meteorological observations, weather forecasting and seismology. IMD is headquartered in Delhi and operates hundreds of observation stations across India and Antarctica. Regional offices are at Mumbai, Kolkata, Nagpur and Pune.

It also issues weather alerts, cyclone alerts for the whole country. IMD is also one of the six Regional Specialised Meteorological Centres of the World Meteorological Organization.
The Surface Instrumentation Division of IMD is responsible for designing, manufacturing, deployment and maintenance of various sensors and weather systems that are used to measure surface meteorological parameters.

IMD in collaboration with ISRO uses IRS series and the Indian National Satellite System (INSAT) for weather monitoring of the Indian subcontinent, making it the first weather bureau of a developing country to develop and maintain its own satellite system.

IMD has continuously ventured into new areas of application and services, and steadily built upon its infrastructure in its history of 140 years. It has simultaneously nurtured the growth of meteorology and atmospheric sciences in India. Today, meteorology in India is poised at the threshold of an exciting future.



MAIN TEXT

Cloud cover is a very important parameter in meteorology, which helps provide a good estimate of real time weather. Many meteorological predictions and decisions rely on this. Hence, a good cloud cover estimation software is needed to provide real-time statistics about the weather.

Although cloud cover can be estimated and reported in percentage, one of the standard unit used a lot by meteorologists is called Okta.

1. OKTA Measurement unit

Okta is a standard measurement unit in meteorology. It gives an estimate of the cloud cover in an area. The range of the Okta scale is from 0 to 8 (Fig. 1.1), where 0 indicates clear skies and 8 denotes completely covered by clouds (Fig. 1.2).

To obtain the cloud cover in okta units, we can scale down the percentage value into 8 integer units.

(a) (b)

Fig 1.2 (a) Clear image of okta 0 (b) Overcast image of okta 8

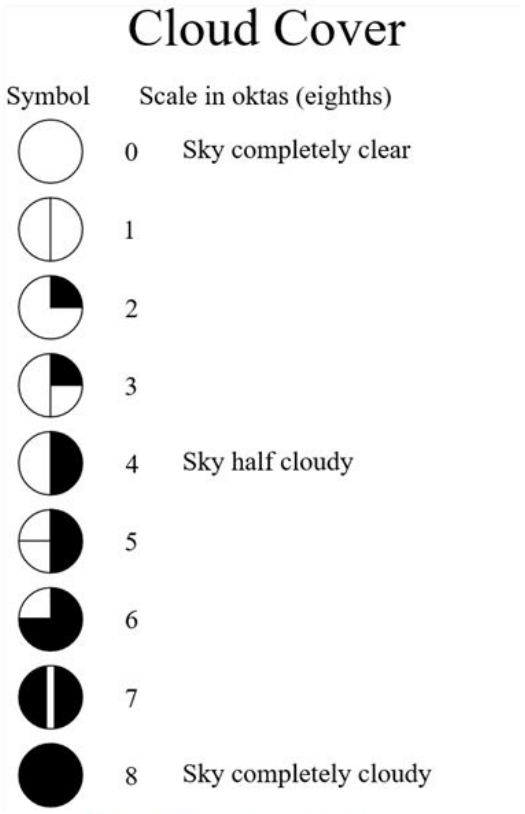


Fig. 1.1 Okta unit and its diagrams

2. Project Work Plan

The work required to obtain cloud cover of an area can be divided into 4 major steps (Fig. 2.1):-

- Take a VIS spectrum image of the sky.
- Perform Image Segmentation on the image, dividing each pixel into cloud or a sky
- Find the percentage of cloud pixels in the image.
- Scale the percentage down to Okta unit

Fig. 2.1 General Workflow for the project

3. Implementations

We found several different ways in which the project could be carried out. These are:

- Using satellite images in IR spectrum, from the official website of the Indian Meteorological Department and calculating cloud cover using thresholding techniques. (Fig. 3.1).
- Using ground images (Fig. 3.2):
    - By applying fixed thresholding on the ground images.
    - By building our own Convolutional Neural Network architecture and training it to perform semantic segmentation on the ground images.
    - By using a pre-defined Deep Learning model (UNET) and training it to perform semantic segmentation.

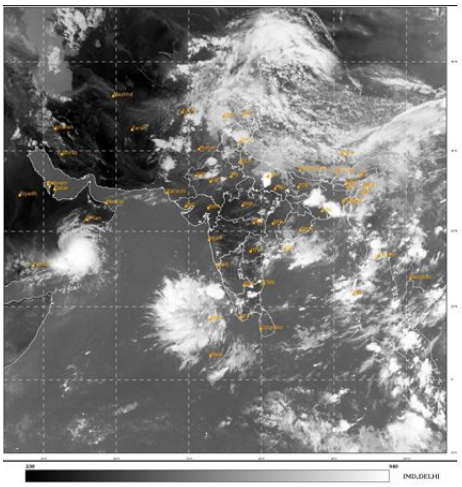


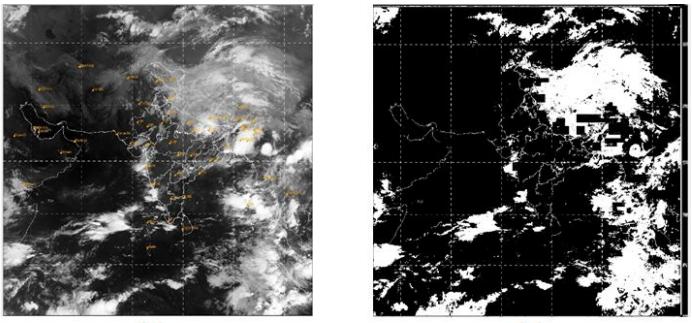
Fig. 3.1 Satellite image from IMD in IR channel (Taken on 30th May)

Fig. 3.2 Ground Visible image of Clouds

4. Using Satellite Images

The Indian Meteorological Department provides satellite images of India in the IR channel on their website. (https://mausam.imd.gov.in/)
We can use this data to calculate the cloud coverage in an area by using simple thresholding techniques. IR images have the benefit that they can be used to segment clouds even in the night time, unlike visible light images. During the night time, visible light images are very hard to segment as the clouds and skies are basically indistinguishable in visible light.
In these satellite images (Fig. 4.1(a)), the intensity of cloud coverage on the map is shown in the range of black to white where black shows visible ground and white denotes clouds. We observe that in the satellite image (Fig.3.1), there is a color legend at the bottom which denotes the range of different clouds for varying gray value. To find the appropriate gray value to perform thresholding, we can use the gray value just at the center of the legend. Values darker than the set gray value would be considered as ground while values above it would be considered as cloud.

Using an image processing operation known as thresholding (*Appendix 1*), we can assign a fixed range of gray values as cloud and the rest as ground. This range is determined by the threshold value we set, above which all the gray values are considered to be clouds, and below which the gray values are classified as ground pixels. We then obtain a map of black and white pixels as in Fig. 4.1(b).



(a) (b)

Fig. 4.1 (a) Satellite image from IMD in IR Channel (Input Image)
(b) Image after applying thresholding

The next step is to map the pixels of the image to the coordinates of the cities. This can be done through quadratic regression (*Appendix 2)*, which is a mathematical formulation to find the equation of a parabola to best fit a set of data points. We use this instead of simple linear regression due to curvature of the Earth, which introduces non linearities in the measurement.
We therefore obtain a list of city names and their corresponding pixel locations in the image. Thus, we can calculate the cloud coverage over any city by considering a fixed area around the city's pixel location.

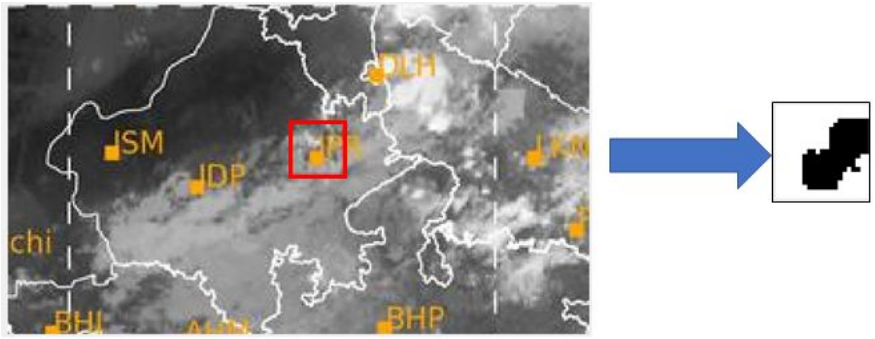


Fig. 4.2 Region considered around the given city to find the cloud cover

Using these ideas, a script is created in MATLAB which could provide the cloud cover of a city required. To be able to run the script, one needs to have a MATLAB licence with Computer Vision toolbox and Deep learning toolbox

4.1. Explanation of the code

- The satellite image from the IMD website is retrieved and the image is converted from RGB to HSV color space.

```
function [cent,okta] = getOkta(y,x)
    % Getting the Image from IMD Site
    img=webread("https://mausam.imd.gov.in/Satellite/3Dasiasec_ir1.jpg");

    %Convert the rgb image to hsv channel
    I = rgb2hsv(img);
    % X is LONGITUTE
    % Y is LATITUDE
```

- After we obtain the image in HSV color space, we apply the mask on the image segment the clouds. The exact values of the channels are obtained from experimental observations.

```
    % Consider only the given range inside to be clouds, rest is ground
    mask = (I(:,:,1) >= 0.000 ) & (I(:,:,1) <= 0.987) & ...
        (I(:,:,2) >= 0.000 ) & (I(:,:,2) <= 0.324) & ...
        (I(:,:,3) >= 0.342 ) & (I(:,:,3) <= 1);
```

- We now have the image where black pixels correspond to ground and white pixels correspond to clouds. Now we have the name of the city to find its cloud cover, we need to find its coordinates to map it into corresponding pixels on the image.

```
    %Find the city and get its coordinates
    if ~exist('x','var')
        city=y;
        cities=getCities('cities.csv');
        if(size(cities(cities.city_name==city,:),1)<1)
            cent=0;okta=0;
            disp("City not Found");
            return
        end
        x=cities(cities.city_name==city,:).longitude;
        y=cities(cities.city_name==city,:).latitude;
    end
```

- Now that we have the coordinates, we map it into the corresponding pixel value. We used quadratic regression (*Appendix 2)* to fit our coordinate system into the correct pixel of the image.

```
28.     %Change to coordinates to pixel values
29.     newy=round(1112.285714-18.96714286*y-0.07357142857*y^2);
30.     newx=round(-899.428571+20.273929*x-8.29857e-4*x^2);
31.
```

- The variable newx and newy are the pixel values that point to the city in our image. Now we take a portion of the image around the city to find the percentage of cloud cover. A region of 20x20 pixels is considered to find the cloud cover of the city.

```
32.     %Crop the image surrounding the given city to find cloud cover
33.     cityimg=mask(newy-10:newy+10,newx-10:newx+10);
34.
```

- Now once we have cropped image of the city, we find out the percentage of white pixels in the image. This would give us the percentage of cloud cover around the city. Then to obtain the value in Okta unit, we scale the percentage down to a scale of 8 units and report the value.

```
35.     %Find the percentage of white pixels and scale it down to okta units
36.     cent=length(cityimg(cityimg==1))/length(cityimg(:)) *100;
37.     okta=round(cent*0.08);
38.
39. end
```

4.2. Limitations

Although this method can work 24x7, Two major problems are seen in this method:

- The result obtained is not in real time
- There are labels and grid mentioned in the image which hampers the correct value of cloud cover

This can be rectified if we are able to obtain a satellite image without any grid or city labels.

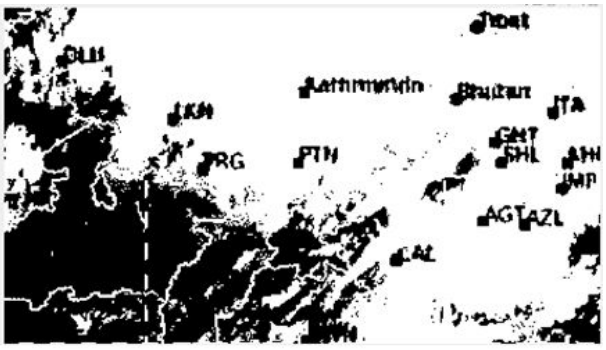


Fig. 4.8 City labels gets identified as ground with thresholding

5. Using Fixed Thresholding on Ground Images

In this method, we use simple fixed thresholding methods to segment ground images, clicked in visible light. The benefits of this is that we don't need Satellite images or IR cameras to perform these operations. We can click pictures of the sky anywhere, using our own cameras and use this technique to calculate the cloud cover. Unfortunately this method does not give good results with night time images of clouds.

Fig.5.1 Different ground images with varying cloud cover

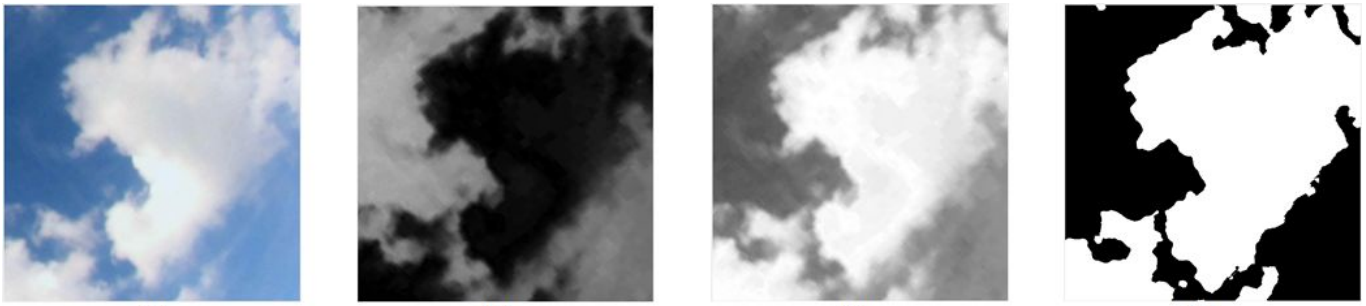
In this method, we take ground images of clouds and apply fixed thresholding (*Appendix 1*) to it, to segment cloud and non-cloud pixels.
There are different ways we can perform the fixed thresholding operation. These are:

- Using Saturation in HSV colorspace: Classifying as cloud or non-cloud pixel based on saturation value of pixel in HSV colorspace (*Appendix 3*). The HSV colorspace is a color model which consists of 3 channels, where instead of the normal RGB channels we use Hue, Saturation and Value to determine the color of a pixel.
- Red-Blue difference: Classification based on absolute difference between Red and blue levels of a pixel.
- Red-Blue ratio: Classification based on ratio of red and blue levels of a pixel.
- Red-Blue normalized: Classification based on the feature (B-R)/(R+B), where 'R' and 'B' denote the red and blue values of a pixel respectively.

For our purposes, we chose to perform the thresholding using the saturation values in HSV colorspace. This is because it serves our needs better as clouds can be easily distinguished by their saturation values. Pure white has a saturation of 0, thus the saturation channel of the HSV color space helps us to differentiate the clouds well, since their saturation will be less compared to sky.

Hence, for our implementation we first convert our image to the HSV colorspace, and then extract the Saturation channel from it so that we can easily differentiate between cloud and sky.

Now, the lower saturation value corresponds to white clouds, thus the darker regions correspond to clouds. This is clearly counter intuitive, hence we invert the image. After this, we get a grayscale image, which we can then apply fixed thresholding on.



(a) (b) (c) (d)

Fig. 5.2. (a) Ground Image, (b) Saturation Channel of image, (c) Inverted Saturation, (d) After Thresholding

Finally, we have a Black and White image, as shown, where black corresponds to sky and white corresponds to cloud regions. From this we can simply calculate the percentage of cloud coverage by dividing the number of white pixels by the total number of pixels. After this we scale down the percentage to Okta units which is between 0-8.

5.1. Explanation of the code

The following is the main MATLAB code which performs these operations.

- The function takes the input image as an argument and returns the binary map of 0s and 1s denoting cloud and non-cloud regions (Fig.5.3.(a)).

```
function [bin] = saturateinv(img)
```

- We convert the image to HSV color space values to obtain the values of the saturation channel (Fig.5.3.(b)).

```
hsv=rgb2hsv(img);    %Convert to HSV color space
gray=hsv(:,:,2);    % 2nd Channel is Saturation channel
```

- Since the white pixels of the cloud have lower saturation than sky pixels, blacker pixels correspond to clouds. Hence we invert the image to get the suitable white pixels as clouds. (Fig.5.3.(c))

```
gray=1-gray;          % Invert Saturation
```

- The light pixels now correspond to clouds while the darker pixels correspond to sky. We binarize the image using Otsu's method and we obtain our desired result of the mask (Fig.5.3.(d)).

```
bin=imbinarize(gray);    % Threshold
```

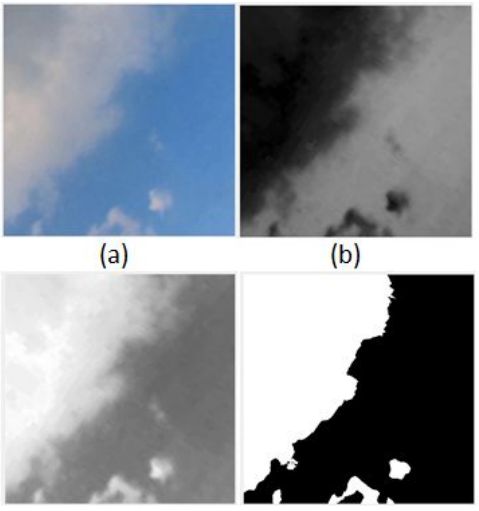


Fig. 5.3. (a) Input Image
(b) Saturation channel from HSV color space
(c) Inversion
(d) Image Segmentation

- Now we can find the percentage of white pixels in the image and scale the value down to okta units. (Fig.5.4)

```
cent=length(bin(bin==1))/length(bin(:)) * 100;
okta=round(cent*0.08);
```

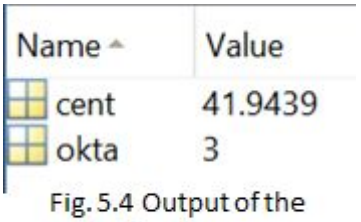
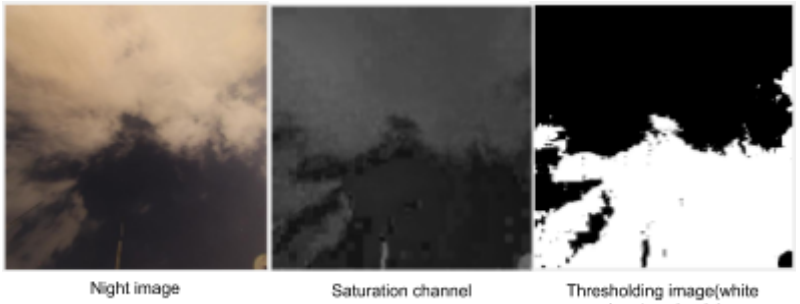


Fig. 5.4 Output of the function

5.2. Limitations

This method helps in producing real time value of cloud cover, but it introduces other errors :-

- This method works only for daytime clouds. In images of night time clouds, the saturation values of the cloud and sky pixels are very close, thus this often causes errors in measurement.
- Dark clouds have a higher value of saturation and hence get detected as sky leading to errors in the value.



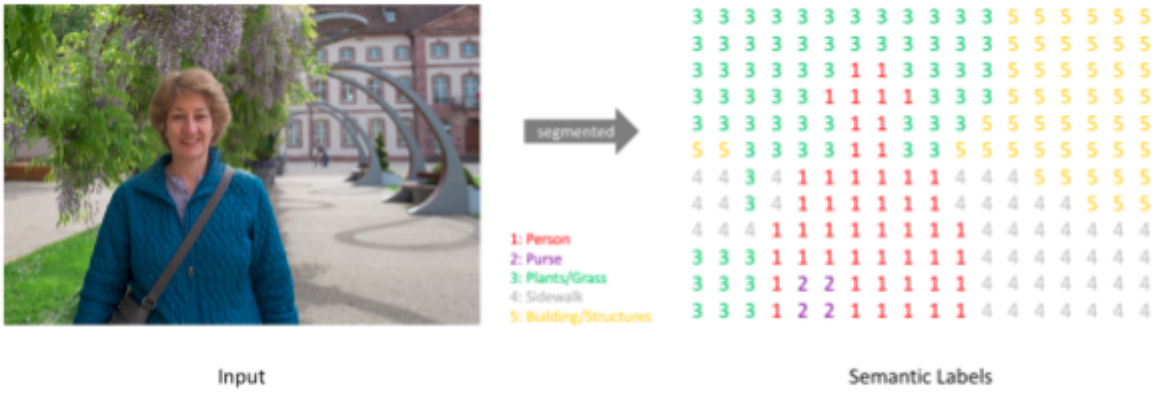
An example of fixed thresholding classifying cloud and sky pixels wrongly

6. Building a Convolutional Neural Network for Semantic Segmentation

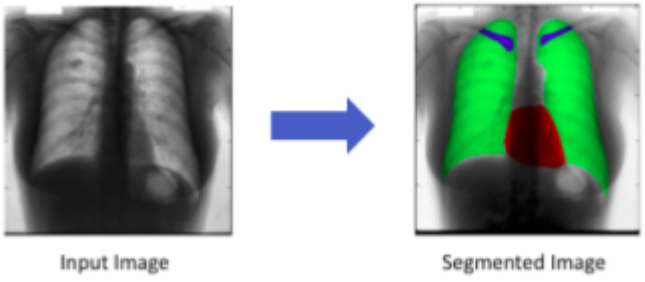
Another implementation of this project can be done using Deep Learning (*Appendix 4)* models to perform semantic segmentation.

6.1. Semantic Segmentation

Image segmentation is a computer vision task in which we label specific regions of an image according to what's present in an image. Specifically, the goal of semantic image segmentation is to label each pixel of an image with a corresponding class of what is being represented.
A neural network performing semantic segmentation takes as input an image, and outputs a segmentation map where every pixel is classified into different classes. This task is very important in tasks like medical imaging.

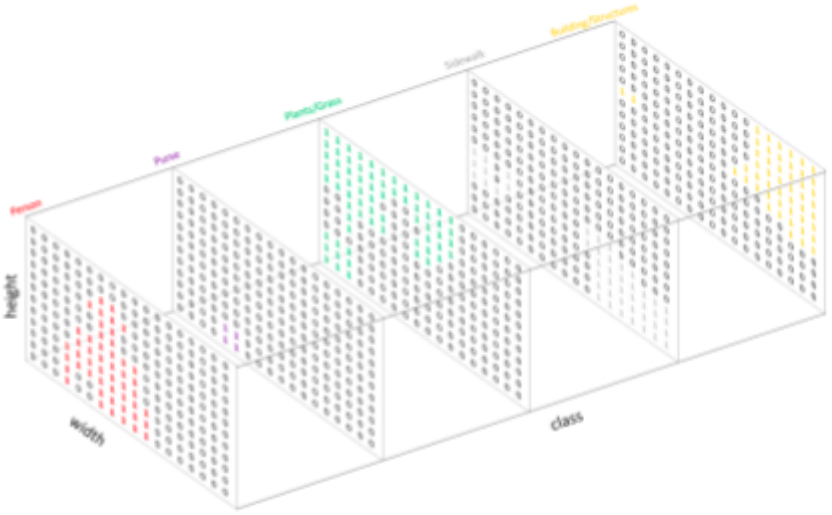


The process of semantic segmentation



Use of semantic segmentation in medical imaging

For our project, we just have 2 classes- cloud and non-cloud. The goal of our neural network is to take a RGB or grayscale image, and output a segmentation map where each pixel contains a class label. We can do this by creating an output channel for every class. Thus we get separate ‘masks’ for every channel, which illuminates the regions where an object of the class is present.



Different output channels corresponding to different classes

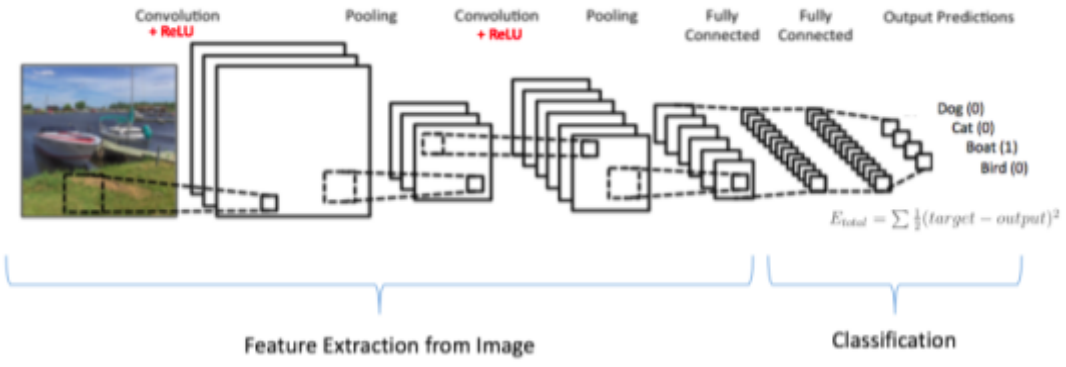
6.2. Convolutional Neural Network

Convolutional Neural Networks are a good choice for semantic segmentation, compared to simple fully connected neural networks, since we are working with 2-D images and it would require a lot of computations for a simple deep network to process a single image. Also, convolutional networks maintain spatial relations between different objects in the image, as it does not flatten out the image into a 1-D array like a simple deep network. This is important in our project.

The input for our neural network would be an RGB image while the output would be the binarized image of the same size with each pixel either black or white denoting cloud or sky.

All CNNs are composed of some fundamental layers, which when arranged differently, give different architectures. These layers are:

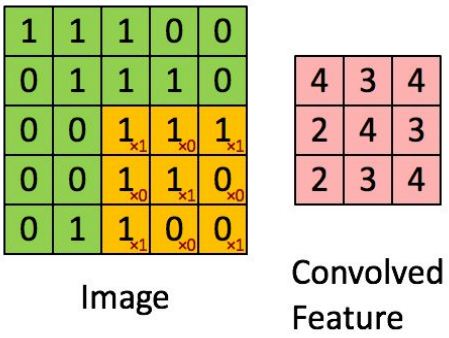
- Convolutional layer
- Pooling layer
- Fully Connected layer
- Relu and Softmax layers
- Transposed Convolution layer



A common CNN architecture

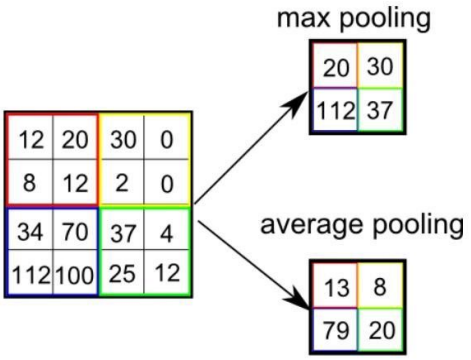
- Convolutional Layer

The objective of the Convolution Operation is to **extract high-level features** such as edges, from the input image. ConvNets need not be limited to only one Convolutional Layer. Conventionally, the initial convolutional layers are responsible for capturing low-level features such as edges, color, gradient orientation, etc. With added layers, the architecture adapts to the High-Level features as well, giving us a network which has a wholesome understanding of images in the dataset, similar to how we would. This layer has a certain number of filters, which are applied to the input image, using the convolution operation. The elements under the view of the kernel are multiplied with its weights, and then summed. This gives the convolved feature.



- Pooling Layer

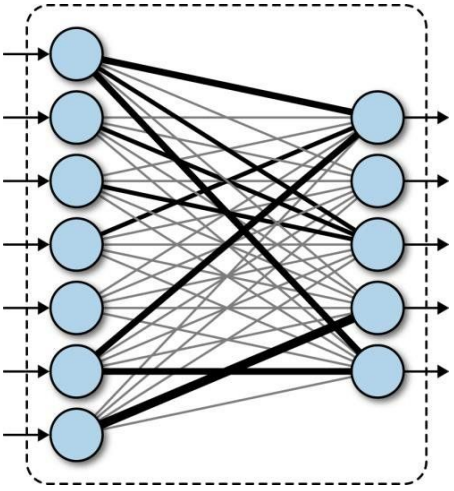
The Pooling layer is responsible for reducing the spatial size of the Convolved Feature. This is to **decrease the computational power required to process the data** through dimensionality reduction. It retains the important features from the convolutional layer. There are three most common types of Pooling - Max, Sum and Average. Max Pooling also performs as a **Noise Suppressant**. It discards the noisy activations altogether and also performs de-noising along with dimensionality reduction. Hence, Max-pooling is usually the standard choice for a pooling layer.



- Fully Connected Layers

After converting the input image into a suitable form, the image is flattened into a column vector. The flattened output is fed to a feed-forward neural network and backpropagation applied to every iteration of training. Over a series of epochs, the model is able to distinguish between dominating and certain low-level features in images and classify them using the **Softmax Classification** technique. The output from the convolutional and pooling layers represent

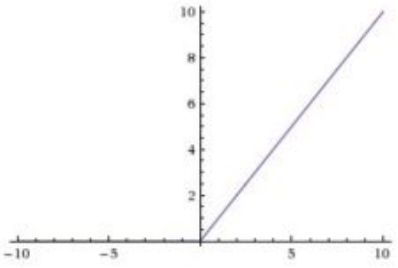
high-level features of the input image. The purpose of the Fully Connected layer is to use these features for classifying the input image into various classes based on the training dataset.



- ReLu

ReLu stands for Rectified Linear Unit. It is a **non-linear activation function** which is introduced after every convolution operation. Activation functions are important as they introduce non-linearities in the functions learned by a neural network. These non-linearities are essential to the working of a Deep learning network as they enable the network to learn more complex functions, compared to functions which are just linear combinations of the input features. ReLU is an element wise operation (applied per pixel) and replaces all negative pixel values in the feature map by zero. There are other activation functions such as tanh, sigmoid and leaky ReLu which aren't as common.

Output = Max(zero, Input)



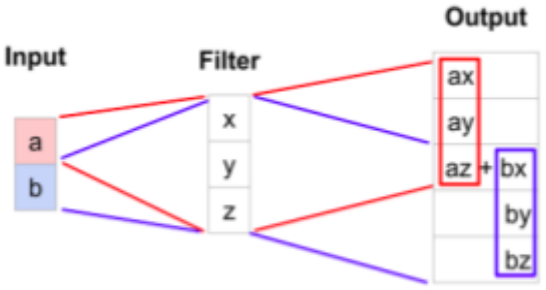
- Softmax Layer

A softmax layer allows the neural network to perform multi-class classification. The neural network, after applying softmax, is able to assign probabilities of an input belonging to different classes. Thus, the outputs of the Softmax Layer sum to 1. The formula for Softmax is:

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

- Transposed Convolutional layer

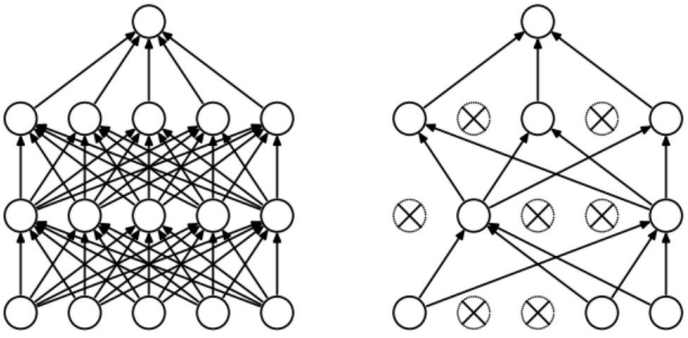
This layer is used for upsampling the image, which has been downsampled throughout the network using successive convolutional and pooling layers. This is important in image segmentation, as we need a segmented map of the same dimensions as the input image. After learning the features through these layers, we need to scale the map up to the image resolution. This is usually done through the transposed convolutional layer, which provides a way for performing **learned upsampling**. It uses a filter to upsample, and thus its weights can be learned through training. For a transpose convolution, we take a single value from the low-resolution feature map and multiply all of the weights in our filter by this value, projecting those weighted values into the output feature map.



Simple 1D example of upsampling using a transposed convolutional layer

- Dropout layer (Optional)

When we train a neural network, there is always a possibility of fitting the training set very well, but failing on inputs which are not part of the training set. This is called overfitting the training set. To prevent this, a technique called regularization is used. There are several different ways of carrying out regularization, one of which is called Dropout. When a network is in the training process, applying dropout takes out some nodes from layers of the network, to prevent the output from being strongly dependent on the features learned from one particular node. This is done by assigning a dropout probability to every layer. This number is the probability of a node from the layer being deactivated in a particular iteration in training.



(a) Standard Neural Net (b) After applying dropout.

The convolutional, max-pooling and transposed convolutional layers have certain parameters which are important to their functioning. These parameters are the stride and padding of the layer, which determines how the filter acts on the input to produce an output.

- Stride

Stride denotes how many values are traversed per slide. A stride of 1 means that the filter traverses through the image one pixel at a time, whereas a stride of 2 means we move the filter 2 pixels at a time. The value of stride greater than 2 results in reduction of the convolved output.

- Padding

On applying convolution to images, we tend to lose out on perimeter pixels. Using convolution in succession would result in a huge loss of outer pixels and we want our model to have the same dimensions as our input since we are classifying pixels. Hence we introduce zeros to the outer boundary of the image, so that the output after convolution remains the same size as of the input image.

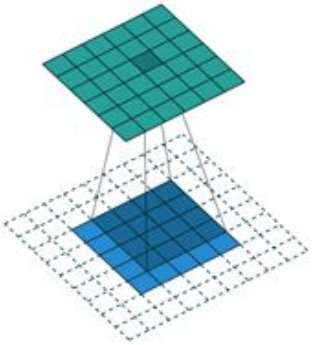


Fig. 6.1 Zero padding making the dimensions consistent

6.3. Our CNN Model

We know that in Deep Learning models, earlier convolutional layers learn low-level features while later layers learn high-level ones. Thus, to learn these features well, we usually need to increase the number of convolutional channels as we get deeper.

One popular approach for image segmentation models is to follow an encoder/decoder structure where we first downsample the spatial resolution of the input, developing lower-resolution feature mappings which are learned to be very efficient at discriminating between classes, and then upsample the feature maps into a full-resolution segmentation map.

To create our CNN model, we used the following architecture,

```
1   ''   Image Input                600x600x3 images with 'zerocenter' normalization
2   ''   Convolution                20 3x3 convolutions with stride [1  1] and padding [1  1  1  1]
3   ''   ReLU                       ReLU
4   ''   Max Pooling                2x2 max pooling with stride [2  2] and padding [0  0  0  0]
5   ''   Convolution                20 3x3 convolutions with stride [1  1] and padding [1  1  1  1]
6   ''   ReLU                       ReLU
7   ''   Transposed Convolution     20 4x4 transposed convolutions with stride [2  2] and cropping [1  1  1  1]
8   ''   Convolution                2 1x1 convolutions with stride [1  1] and padding [0  0  0  0]
9   ''   Softmax                    softmax
10  ''   Pixel Classification Layer Cross-entropy loss
```

6.4. Dataset

We had a dataset called SWIMSEG which is used to train our neural network

- 1013 images of resolution 600x600
- Features ground truth of all the images
- Has a variety of cloud types
- Images taken at daytime only.

Out of these 1013 images, we divided the dataset as follows:

Training - 900 images
Testing - 100 images

6.5. Explanation of the code

Our model has been created in MATLAB which takes care of setting parameters and backpropagation on its own. To run this script, a MATLAB licence is required along with Computer Vision and Deep Learning toolbox.

- ### Training

  - First we create 2 datastores for our image and ground truth images. We also denote that white pixels in ground truth images correspond to cloud and black ones to the sky.

```
%% Load Image dataset

dataSetDir=fullfile("C:\Users\KIIT\Downloads\imgs\cloudImages\swimseg");
imageDir=fullfile(dataSetDir,'images');
labelDir=fullfile(dataSetDir,'GTmaps');
imds=imageDatastore(imageDir);   %Datastore of VIS Images
labelIDs=[0 255];classNames=["sky" "clouds"];
pxds=pixelLabelDatastore(labelDir,classNames,labelIDs); %Datastore of GT
```

  - Next, we create our layers for our Neural Network to train

```
%% Create CNN

numFilters=20;
filterSize=3;
numClasses=2;
layers=[
    imageInputLayer([600 600 3])
    convolution2dLayer(filterSize,numFilters,'Padding',1)
    reluLayer()
    maxPooling2dLayer(2,'Stride',2)
    convolution2dLayer(filterSize,numFilters,'Padding',1);
    reluLayer()
    transposedConv2dLayer(4,numFilters,'Stride',2,'Cropping',1);
    convolution2dLayer(1,numClasses);
    softmaxLayer()
    pixelClassificationLayer()
    ];
```

- Once we created the model, we began training our neural network with inputs as our two datastores.

```
%% Training
opts=trainingOptions('sgdm','InitialLearnRate',1e-3,'MaxEpochs',40,'MiniBatchSize',20);
trainingData=pixelLabelImageDatastore(imds,pxds);
net=trainNetwork(trainingData,layers,opts);
```

- MATLAB starts training the model and stops after going around the epochs given.

```
Training on single CPU.
Initializing input data normalization.
|========================================================================================|
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Mini-batch  |  Base Learning  |
|         |             |   (hh:mm:ss)   |   Accuracy   |     Loss     |      Rate       |
|========================================================================================|
|      34 |        1500 |       04:43:00 |       90.70% |       0.2347 |          0.0010 |
|      35 |        1550 |       04:52:03 |       92.65% |       0.2036 |          0.0010 |
|      36 |        1600 |       05:01:09 |       85.28% |       0.3367 |          0.0010 |
|      37 |        1650 |       05:10:17 |       85.54% |       0.3800 |          0.0010 |
|      38 |        1700 |       05:19:20 |       89.53% |       0.2260 |          0.0010 |
|      39 |        1750 |       05:28:28 |       87.16% |       0.2840 |          0.0010 |
|      40 |        1800 |       05:37:32 |       89.40% |       0.2634 |          0.0010 |
|========================================================================================|
>>
```

- Testing

- Once our model is trained, we use our testing dataset to find out the accuracy on our dataset.

Command Window

```
* Data set metrics:

    GlobalAccuracy    MeanAccuracy    MeanIoU    WeightedIoU    MeanBFScore
    ______________    ____________    _______    ___________    ___________

        0.86717           0.86496      0.76404      0.76483        0.51418
```

Fig.6.2. Accuracy and other metrics on our dataset

6.6. Result

The neural network was able to achieve 86% accuracy on the test set, and produced good results on cloud images which were hard to segment using fixed thresholding.

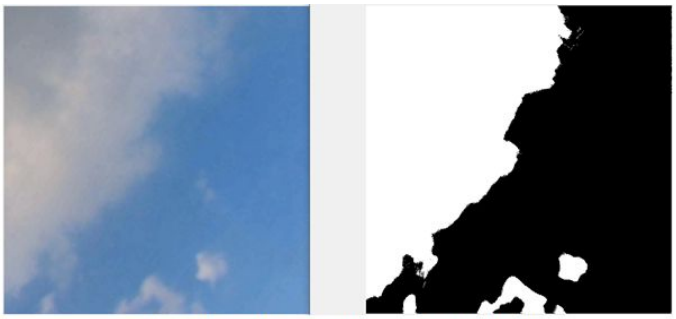


Fig.6.4 Semantic Segmentation of a dark cloud

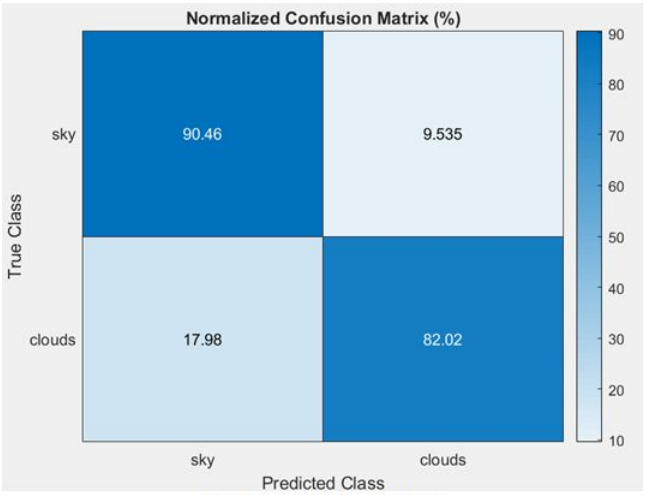


Fig.6.3. Confusion Matrix

6.7. Limitations

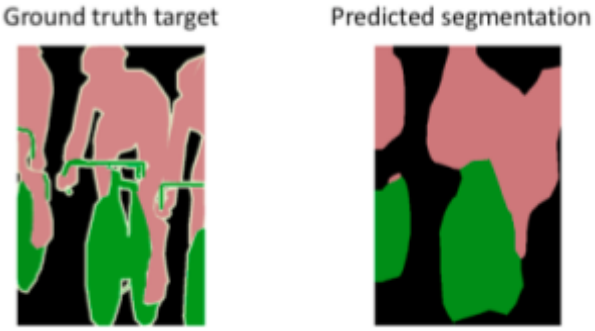
This implementation, like the fixed thresholding one, has the drawback of not being useful in classifying the images of clouds in the night sky. The clouds and sky are very hard to distinguish in the night sky, in visible light, and hence the model cannot be used in such situations. Also, since the SWIMSEG dataset is a relatively small dataset, the model is overfitting the training data and struggling to classify real life images. Wavy clouds, in particular, seem to be providing inaccuracies.

Inaccurate results on images of wavy clouds

7. Using Pre-Defined Architecture for Training (UNET)

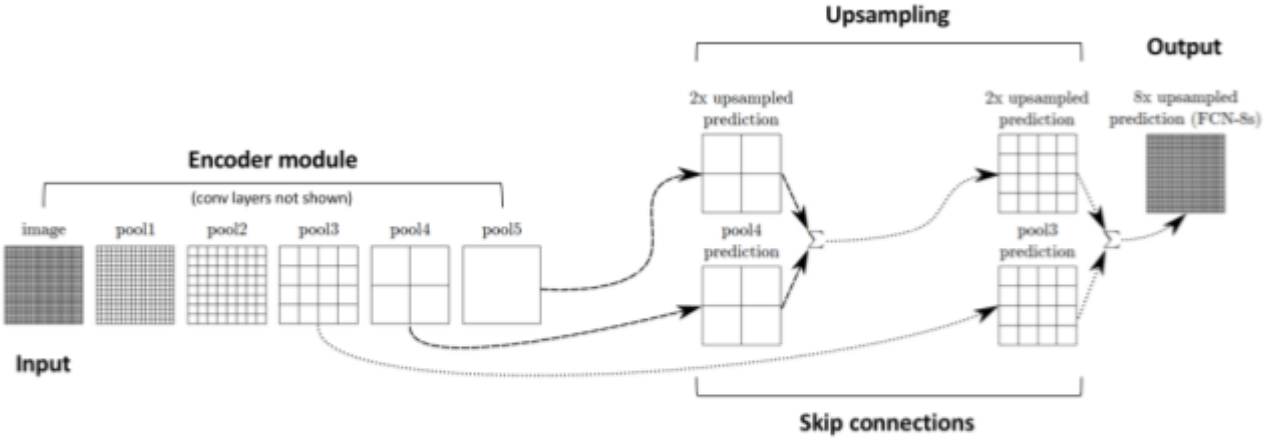
There are several common models in the world of Deep Learning, that have been built through extensive research. These models are built for specific tasks. Some popular architectures are LeNet5, AlexNet, ResNet, UNET etc. Amongst these, UNET is an architecture that was specifically developed for image segmentation tasks in the field of medical imaging.

Convolutional Neural Networks give good results on easier image segmentation problems but it hasn't made any good progress on complex ones. This is mainly because the convolutions and max-pooling layers reduce spatial dimensions by a big factor, and thus the final few layers of transposed convolutions struggle to produce fine-grained segmentation maps. The maps obtained do not have clear boundaries separating the classes.



ConvNets struggle to produce clear boundaries in the segmentation maps

This problem can be solved by upsampling slowly in sections, and adding "skip connections" from earlier layers, and later summing these two feature maps. These connections are from previous layers in the network, before the downsampling, so they help to provide the detail needed to reconstruct the boundaries in the segmentation maps.



Working of skip connections. Earlier layers are added to the upsampling, to provide detail to reconstruct boundaries.

7.1. UNET

UNET is a Neural Network architecture that was originally developed for medical imaging purposes. It uses the ideas of upsampling slowly, and using skip connections to provide a detailed reconstruction of the object boundaries in the input image. As shown in Fig 7.1, it looks like a ‘U’, which justifies its name.

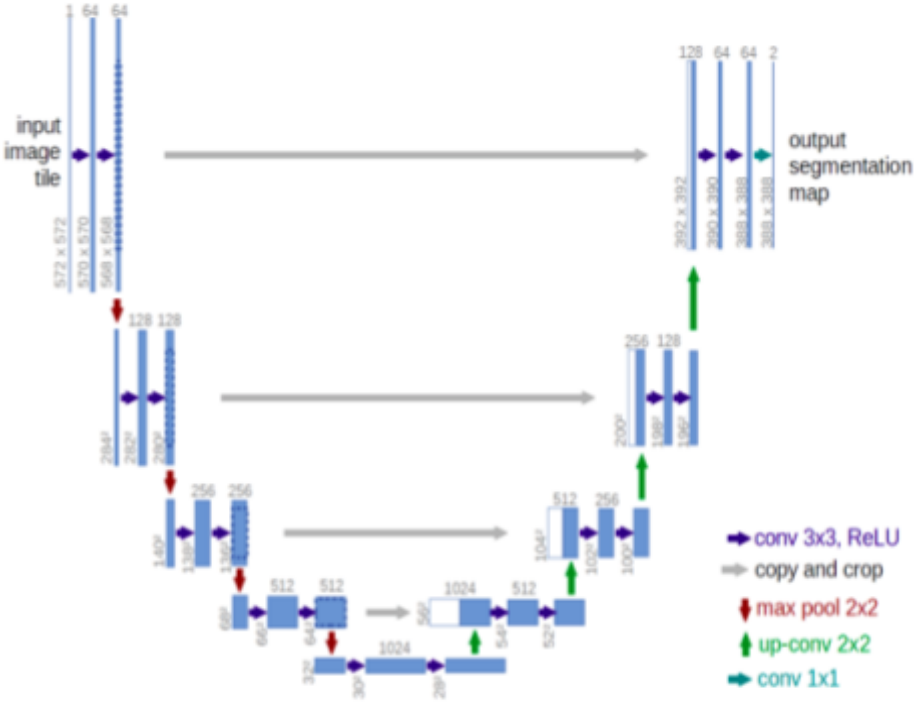


Fig 7.1: The UNET architecture

(Note: The original UNET architecture used valid padding, instead of same padding, so the spatial dimensions are reduced by 2 after every convolution operation in the image of the model above)

This architecture has three sections. These are, the **downsampling or contracting path**, **the bottleneck section** and the **upsampling or expanding path**.

- The contracting path is made of several contraction blocks. These blocks are composed of two 3X3 convolutions, followed by a 2X2 max pooling layer. Throughout the contracting path, the number of kernels, or feature maps doubles after each block, so that the neural network can learn more complex features more effectively. Also, after every block, the width and height of the input image becomes half.
- The bottleneck section is an intermediary between the contracting and expanding path. It consists of two 2X2 convolutions applied in succession, followed by a 2X2 up-convolutional layer which uses transposed convolution to start the upsampling path.
- After the bottleneck section, the expanding path starts the upsampling process by slowly increasing the spatial dimensions of the downsampled input image. The upsampling path is the main feature of this architecture, as it uses skip connections from the previous layers as shown in FIg 7.1. The expanding path is symmetric to the contracting path, and has the same number of expansion blocks as the contracting path has contraction blocks. Each of these blocks passes the input through two 2X2 convolutional layers, followed by an up-convolutional layer. But, before doing these operations, the input is appended by the corresponding map from the downsampling section using a skip

connection. This ensures that details of edges are not lost during upsampling. Consecutive expanding blocks progressively half the number of filters, because the goal is to obtain a segmentation map which, for the purposes of this project, has just one channel. Also, throughout the expanding path, the width and height of the input image doubles in every block.

- Finally, the obtained result is passed through a 1X1 convolutional layer, with the number of feature maps equal to the number of different segments we need. For the purposes of this project, just one feature map suffices, since the goal is to only classify cloud and non-cloud pixels.

7.2. Defining the UNET model

Using the UNET architecture, and fixing parameters like input image size, number of kernels in layers, stride and padding of layers, the model defined is as follows:

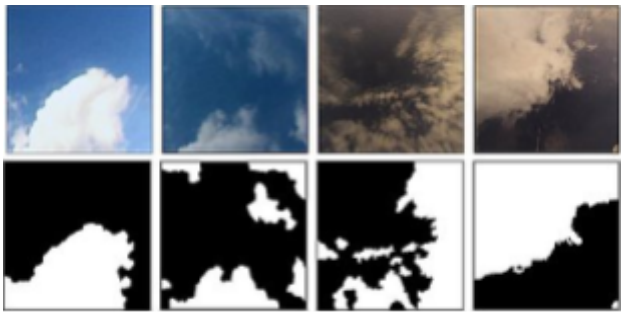
Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, 128, 128, 3)	0	
conv2d (Conv2D)	(None, 128, 128, 16)	448	input_1[0][0]
conv2d_1 (Conv2D)	(None, 128, 128, 16)	2320	conv2d[0][0]
max_pooling2d (MaxPooling2D)	(None, 64, 64, 16)	0	conv2d_1[0][0]
dropout (Dropout)	(None, 64, 64, 16)	0	max_pooling2d[0][0]
conv2d_2 (Conv2D)	(None, 64, 64, 32)	4640	dropout[0][0]
conv2d_3 (Conv2D)	(None, 64, 64, 32)	9248	conv2d_2[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 32, 32, 32)	0	conv2d_3[0][0]
dropout_1 (Dropout)	(None, 32, 32, 32)	0	max_pooling2d_1[0][0]
conv2d_4 (Conv2D)	(None, 32, 32, 64)	18496	dropout_1[0][0]
conv2d_5 (Conv2D)	(None, 32, 32, 64)	36928	conv2d_4[0][0]
max_pooling2d_2 (MaxPooling2D)	(None, 16, 16, 64)	0	conv2d_5[0][0]
dropout_2 (Dropout)	(None, 16, 16, 64)	0	max_pooling2d_2[0][0]
conv2d_6 (Conv2D)	(None, 16, 16, 128)	73856	dropout_2[0][0]
conv2d_7 (Conv2D)	(None, 16, 16, 128)	147584	conv2d_6[0][0]
max_pooling2d_3 (MaxPooling2D)	(None, 8, 8, 128)	0	conv2d_7[0][0]
dropout_3 (Dropout)	(None, 8, 8, 128)	0	max_pooling2d_3[0][0]
conv2d_8 (Conv2D)	(None, 8, 8, 256)	295168	dropout_3[0][0]
conv2d_9 (Conv2D)	(None, 8, 8, 256)	590080	conv2d_8[0][0]
up_sampling2d (UpSampling2D)	(None, 16, 16, 256)	0	conv2d_9[0][0]
concatenate (Concatenate)	(None, 16, 16, 384)	0	up_sampling2d[0][0] conv2d_7[0][0]
dropout_4 (Dropout)	(None, 16, 16, 384)	0	concatenate[0][0]
conv2d_10 (Conv2D)	(None, 16, 16, 128)	442496	dropout_4[0][0]
conv2d_11 (Conv2D)	(None, 16, 16, 128)	147584	conv2d_10[0][0]
up_sampling2d_1 (UpSampling2D)	(None, 32, 32, 128)	0	conv2d_11[0][0]
concatenate_1 (Concatenate)	(None, 32, 32, 192)	0	up_sampling2d_1[0][0] conv2d_5[0][0]

dropout_5 (Dropout)	(None, 32, 32, 192)	0	concatenate_1[0][0]
conv2d_12 (Conv2D)	(None, 32, 32, 64)	110656	dropout_5[0][0]
conv2d_13 (Conv2D)	(None, 32, 32, 64)	36928	conv2d_12[0][0]
up_sampling2d_2 (UpSampling2D)	(None, 64, 64, 64)	0	conv2d_13[0][0]
concatenate_2 (Concatenate)	(None, 64, 64, 96)	0	up_sampling2d_2[0][0] conv2d_3[0][0]
dropout_6 (Dropout)	(None, 64, 64, 96)	0	concatenate_2[0][0]
conv2d_14 (Conv2D)	(None, 64, 64, 32)	27680	dropout_6[0][0]
conv2d_15 (Conv2D)	(None, 64, 64, 32)	9248	conv2d_14[0][0]
up_sampling2d_3 (UpSampling2D)	(None, 128, 128, 32)	0	conv2d_15[0][0]
concatenate_3 (Concatenate)	(None, 128, 128, 48)	0	up_sampling2d_3[0][0] conv2d_1[0][0]
dropout_7 (Dropout)	(None, 128, 128, 48)	0	concatenate_3[0][0]
conv2d_16 (Conv2D)	(None, 128, 128, 16)	6928	dropout_7[0][0]
conv2d_17 (Conv2D)	(None, 128, 128, 16)	2320	conv2d_16[0][0]
conv2d_18 (Conv2D)	(None, 128, 128, 1)	17	conv2d_17[0][0]

Total params: 1,962,625
Trainable params: 1,962,625
Non-trainable params: 0

7.3. Dataset

For training the network we used the SWINySEG dataset. This dataset has 6768 day and night time images of sky/cloud patches and their corresponding binary segmentation maps. It is obtained from the SWIMSEG dataset mentioned earlier, and a separate dataset called SWINSEG (which has 115 night time cloud images). The images from these 2 datasets were taken and rotated at various angles to give 6 different images for every single image. This is a method of **data augmentation**, which is a method to increase the available training data.



Images from the SWINySEG dataset

For training, the training-validation split was done as follows:

- 6168 images were used for the training set
- 600 images were used for the validation set, upon which the module could be evaluated, for different parameter values.

7.4. Explanation of the code

- First, the contraction and expansion blocks are defined using convolution, max pooling, upsampling and dropout layers, which are predefined in the keras library. These blocks are used in the contracting and expanding paths respectively in the model. (**layers.py**)

    - Contraction block: The contraction block is the block which is repeatedly used in the downsampling process. It takes arguments: **x**, which is the input to the layer; **filters**, which is the number of filters in the block; **dropout rate**, which is the parameter which assigns dropout probability in the block; **kernel size,** which is the dimensions of the convolutional layer; **padding**, which is the type of padding used; and **strides**, which defines the number of strides. This block consists of two convolutions, followed by a max pooling layer. Then Dropout regularization is applied. It returns the convolutional layer output as well, as it is needed for the skip connections in the expansion path.

```
def down_block(x, filters, dropout_rate, kernel_size=(3, 3), padding="same", strides=1):
    conv = keras.layers.Conv2D(filters, kernel_size, padding=padding, strides=strides, activation="relu")(x)
    conv = keras.layers.Conv2D(filters, kernel_size, padding=padding, strides=strides, activation="relu")(conv)
    pool = keras.layers.MaxPool2D((2, 2), (2, 2))(conv)
    pool = keras.layers.Dropout(dropout_rate)(pool)

    return conv, pool
```

    - Bottleneck layer: This is the intermediate layer between the contracting and expanding paths. The arguments have the same meaning as they did in the contraction block. This layer consists of two convolution layers in succession.

```
# Bottlenecking
def bottleneck(x, filters, kernel_size=(3, 3), padding="same", strides=1):
    conv = keras.layers.Conv2D(filters, kernel_size, padding=padding, strides=strides, activation="relu")(x)
    conv = keras.layers.Conv2D(filters, kernel_size, padding=padding, strides=strides, activation="relu")(conv)
    return conv
```

    - Expanding block: This block is the repeating block in the expanding path in the network. The arguments have the same meaning as they did in the contraction block, with the addition of **skip**, which is the layer from where a skip connection is taken, and concatenated. Here, first the input is upsampled using an upsampling layer, which applies transposed convolution to the image. Then the skip connected layer is concatenated, followed by dropout and 2 convolutional layers.

```
def up_block(x, skip, filters, dropout_rate, kernel_size=(3, 3), padding="same", strides=1):
    us = keras.layers.UpSampling2D((2, 2))(x)
    concat = keras.layers.Concatenate()([us, skip])
    concat = keras.layers.Dropout(dropout_rate)(concat)
    conv = keras.layers.Conv2D(filters, kernel_size, padding=padding, strides=strides, activation="relu")(concat)
    conv = keras.layers.Conv2D(filters, kernel_size, padding=padding, strides=strides, activation="relu")(conv)

    return conv
```

- Defining the model(**UNET.py**): Using the blocks defined previously, we construct the UNET architecture. the num_filters variable stores the number of filters for the various blocks, which are constantly doubling. The downsampling path is then defined using 4 contracting blocks which progressively reduces the image dimensions from 128x128 down to 8X8 till the bottleneck layer. Then, the upsampling section is defined using four expanding blocks, and the num_filters used in opposite order. The skip connections are also made using the previous convolutional layer outputs.
  In the end a convolution layer with 1 filter is applied, to give a single channel output segmentation map for our input image.

```
def UNet():

    # number of filters in every block
    num_filters = [16, 32, 64, 128, 256]

    inputs = keras.layers.Input((image_size, image_size, 3))

    pool0 = inputs

    # Downsampling
    #128 -> 64
    conv1, pool1 = down_block(pool0, num_filters[0], input_dropout_rate)

    #64 -> 32
    conv2, pool2 = down_block(pool1, num_filters[1], dropout_rate)

    #32 -> 16
    conv3, pool3 = down_block(pool2, num_filters[2], dropout_rate)

    #16->8
    conv4, pool4 = down_block(pool3, num_filters[3], dropout_rate)


    # Bottlenecking
    botn = bottleneck(pool4, num_filters[4])


    # Upsampling
    #8 -> 16
    uconv1 = up_block(botn, conv4, num_filters[3], dropout_rate)

    #16 -> 32
    uconv2 = up_block(uconv1, conv3, num_filters[2], dropout_rate)

    #32 -> 64
    uconv3 = up_block(uconv2, conv2, num_filters[1], dropout_rate)

    #64 -> 128
    uconv4 = up_block(uconv3, conv1, num_filters[0], dropout_rate)


    outputs = keras.layers.Conv2D(1, (1, 1), padding="same", activation="sigmoid")(uconv4)
    model = keras.models.Model(inputs, outputs)
    return model
```

- Training the model(**train.py**): We first initialise parameters like the image size, batch size and the number of images used in the validation set. We also initialize the parameters for dropout rates of the input layers and the other layers.
  For the program to find the images and ground truths of the dataset, we have to have a Datasets/ directory in the parent directory of the source code files containing the SWINySEG dataset, with subfolders images/ and GTmaps/. This directory structure is set up when the repository is cloned. We then load the ids of the images (ground truth ids are the same) into a list, and divide these ids into training set and validation set ids.

```
# Setting up parameters
image_size = 128
batch_size = 8
val_data_size = 600

# Dropout parameters for regularization
input_dropout_rate = 0.25
dropout_rate = 0.5

current_dir = os.getcwd()
parent_dir = os.path.abspath(os.path.join(current_dir, os.pardir))
# Training path having images and ground truths in different director
train_path = parent_dir + "/Datasets/SWINySEG/"

# Get image, GT ids. SWINySEG has .jpg images and .png GTs. To refer
lst = os.listdir(train_path+"images/")
train_ids = [x.split('.')[0] for x in lst]

# Divide ids into training and validation
valid_ids = train_ids[:val_data_size]
train_ids = train_ids[val_data_size:]
```

We then define the learning rate, loss function and the optimiser we use to train the model. We have used the Adam optimiser and the binary cross-entropy loss function. Then we initialise the training and validation set generator objects. After that we define the number of epochs (the number of iterations through the entire dataset) and finally we train the model using the model.fit() function. Lastly we save the weights obtained from training in the Weights/ directory.

```
# Learning rate for adam
learning_rate = "0.0005"

opt = keras.optimizers.Adam(learning_rate=float(learning_rate))
model.compile(optimizer=opt, loss="binary_crossentropy", metrics=["acc"])
model.summary()


train_gen = DataGen(train_ids, train_path, image_size=image_size, batch_size=batch_size)
valid_gen = DataGen(valid_ids, train_path, image_size=image_size, batch_size=batch_size)

train_steps = len(train_ids)//batch_size
valid_steps = len(valid_ids)//batch_size

# Number of epochs
num_epochs = 200

model.fit(train_gen, validation_data=valid_gen, steps_per_epoch=train_steps, validation_steps=valid_steps,
                    epochs=num_epochs)

# To save weights. Make sure there is a Weights/ directory in the same directory as the program.
model.save_weights(current_dir+"/Weights/"+str(learning_rate)+str(batch_size)+str(input_dropout_rate)+str(dropout_rate)+".h5")
```

- Prediction(**GetOkta.py**): Then we define the functions to make predictions on input images, based on the learned weights.
  The predict() function takes an input image and outputs a binary segmentation map, where the black pixels denote the non-cloud regions and the white pixels show the cloud regions. It does this by applying the model with the pretrained weights on the image, and then classifying all the gray values above 0.5 as non-cloud, and the others as cloud pixels. It plots the image and result side by side.

```
def predict(image, image_name):
    image = np.expand_dims(image, axis=0)
    result = model.predict(image)

    result = result>0.5
    pred = np.reshape(result[0], (image_size, image_size))

    percentage, okta = get_okta(pred)

    fig, (ax1, ax2) = plt.subplots(1, 2)
    fig.suptitle("The cloud cover in image '{}' = {}% OR {} OKTA".format(image_name, percentage, okta))

    ax1.imshow(image[0])
    ax1.set_title("Image")
    ax2.imshow(pred*255, cmap="gray")
    ax2.set_title("Prediction")
    plt.show()

    return pred
```

We then define the get_okta() function which takes a binary segmentation map and returns the cloud coverage in percentage in okta.

```
def get_okta(pred):
    num_pix = image_size*image_size
    cloud_pix = np.count_nonzero(pred==1)

    percentage = cloud_pix*100/num_pix
    okta = round(percentage/12.5)

    return percentage, okta
```

Then we define the path to the images. To test with your own images, put your images in the Test/ directory. Then we compile the ids of the test images in the test_ids list. We also load the pre-trained weights from the Weights/ directory. The weights_filename variable can be changed according to which set of weights one wants to load.

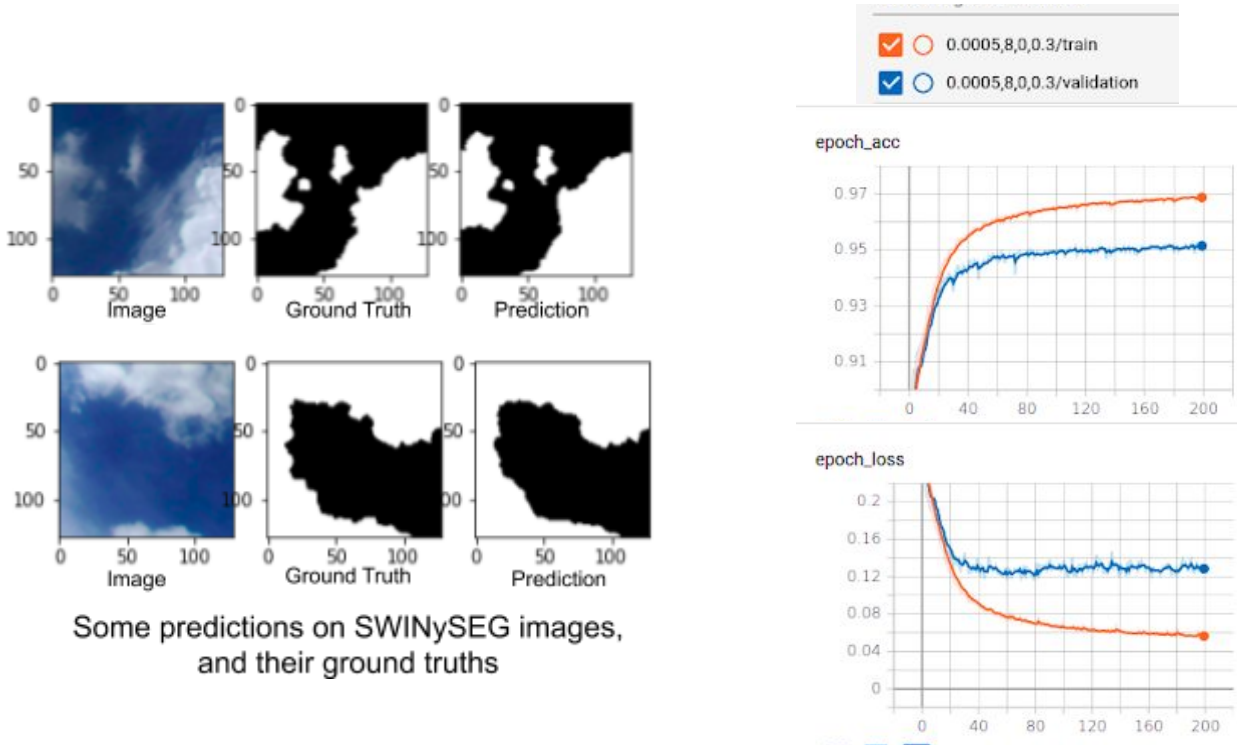
```
# Test path having test images
test_path = current_dir + "/Test/"
# Test image ids
test_ids = next(os.walk(test_path))[2]
# Directory to load trained weights from
weights_dir = current_dir + "/Weights/"
weights_filename = "0.0005800.3.h5"
```

Finally, we load the weights and obtain predictions and cloud coverage values of every image in the Test/ directory by running a for loop through the images, and applying the predict() function on all the images.

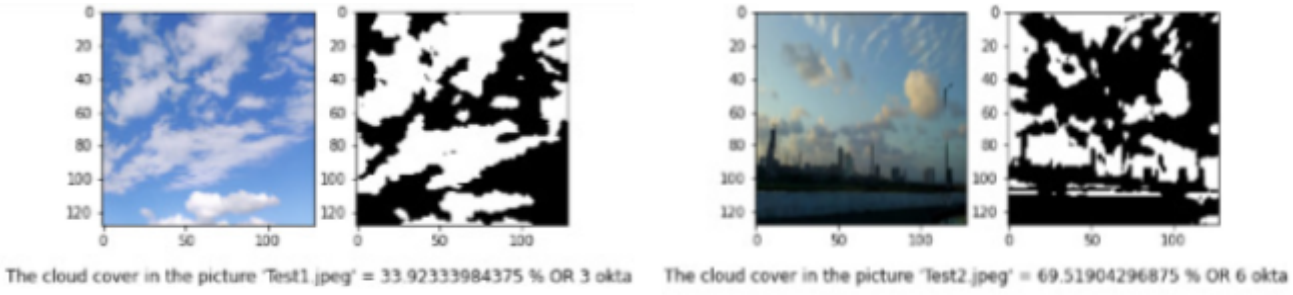
```
# Loading weights
weights = model.load_weights(weights_dir + weights_filename)

for image_id in test_ids:
    image_path = os.path.join(test_path, image_id)
    image = cv2.imread(image_path, 1)
    image = cv2.resize(image, (image_size, image_size))
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    pred = predict(image, image_id)
```

7.5. Result

Our UNET model was able to achieve a epoch training set accuracy of 96.9% and a epoch validation set accuracy of 95.2%.



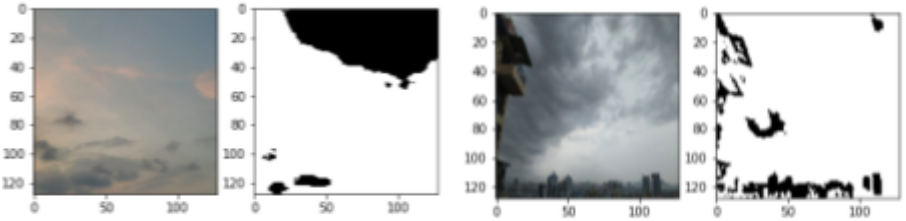
Some predictions on SWINySEG images, and their ground truths



Predictions on non-SWINySEG images

7.6. Limitations

This implementation, too, has the problem of not being able to detect clouds in the night sky. This is realistically only possible with IR images. Also, the model does not perform very well with images of dark clouds, or alternatively, of very light skies.



The model does not perform well with dark clouds or light skies

8. Metrics to Analyze Performance

There are few metrics which can be used to judge how well our network is performing. Just finding accuracy is not sufficient enough and we need to have other variables to see how our predictions fair up with the true cases.

Before explaining the terms we need some definitions. A Confusion matrix shows how well our predictions match with the true cases. Four types of data can be retrieved from it :-

1. True positive (TP) - the model *correctly* predicts the *positive* class.
2. True negative (TN) - model *correctly* predicts the *negative* class.
3. False positive (FP) - model *incorrectly* predicts the *positive* class.
4. False negative (FN) - the model *incorrectly* predicts the *negative* class.

8.1. Accuracy

Accuracy is the fraction of predictions our model got right. It is defined as :-

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} \qquad \text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Accuracy close to 1 denotes that our model is predicting correct values for the pixels, whereas values closer to 0 denotes large deviation in predicting correct values.

8.2. Precision and Recall

- Precision

Precision denotes what proportion of positive predictions were actually correct ? It can be defined as

$$\text{Precision} = \frac{TP}{TP + FP}$$

- Recall

Recall denotes what proportion of actual positives were identified correctly as positive ?

$$\text{Recall} = \frac{TP}{TP + FN}$$

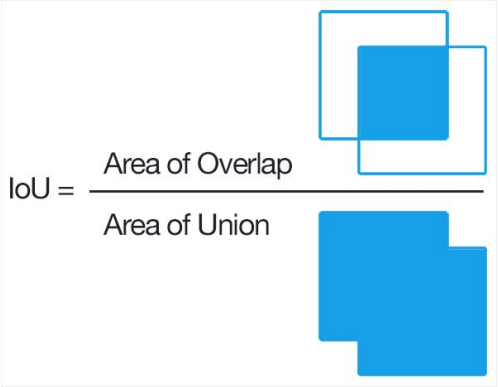
8.3 F1-Score

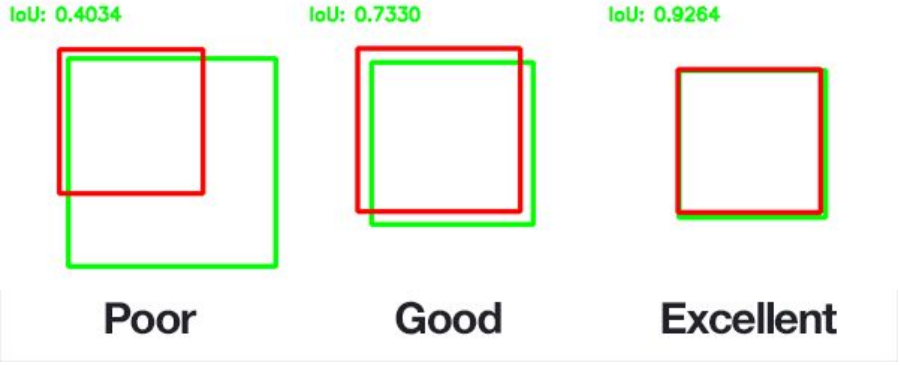
F1-score is simply the harmonic mean of recall and precision. This metric can be used to find optimal value of precision and recall.

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

8.4 Intersection over Union

Intersection over Union is the ratio of the area of intersection of the prediction and true cases to the area of union of the same. Value close to 1 denotes strong correlation between the predictions and ground truth, whereas value close to 0 denotes no correlation in between predictions and true cases.





RESULTS

Through our implementations, we have devised four different methods of calculating cloud coverage over an area. Each of these have different pros and cons and have different areas that could be improved upon.

Using Satellite IR images, one can calculate the cloud coverage at night as well, thus it provides the cloud coverage throughout the day. However, it has the problem that the cloud coverage measurement is not a real-time one.

Using ground images, real-time cloud coverage can be obtained and we devised 3 seperate ways of accomplishing this task. Using simple thresholding operations on these ground images, we obtain decent results for most images, but for darker clouds, this method did not yield promising results.

To improve accuracies, we constructed a Convolutional Neural Network architecture and trained it on a dataset of cloud and sky images. This gave an accuracy of 88%, but seemed to fail with particular types of clouds like wavy clouds.

Another implementation was by using a predefined Neural Network model, UNET, which is a widely used architecture for semantic segmentation purposes. This gave a better accuracy of 96.9 percent on the dataset, but fails with images of dark clouds in light skies.

All of the methods using ground images do not work with images of night skies as the clouds are virtually indistinguishable in visible light during the night time.

Method	Accuracy(%)	IoU	Recall	Precision	F1-Score
Fixed Thresholdidng (Saturation)	91.2776	0.8122	0.9297	0.8931	0.9110
Making our own CNN	88.0847	0.7845	0.8979	0.8595	0.8782
Using Unet Architecture	94.0313	0.8600	0.9349	0.9465	0.9407

Table C.1. Performance of our three implementations

CONCLUSIONS AND RECOMMENDATIONS

We have successfully implemented the above techniques for measuring cloud coverage, but each has its limitations. The method using satellite IR data has its flaws due to the labels and grids in the images provided by IMD. The simple thresholding method and the semantic segmentation method fail with images of clouds taken at night.

Our implementation using Neural networks has thus far achieved an accuracy of 88% in classifying cloud pixels, which was improved by using a UNet architecture giving us an accuracy of 94% in our dataset.

Both our implementations of the Fixed Thresholding method and Semantic Segmentation method can be made to work in the night as well, if we used them with IR images instead of Visible light images. For the neural network, we would need a dataset of IR images to train the network on. This would provide a robust method which can be used to give the cloud coverage throughout the day.

APPENDICES

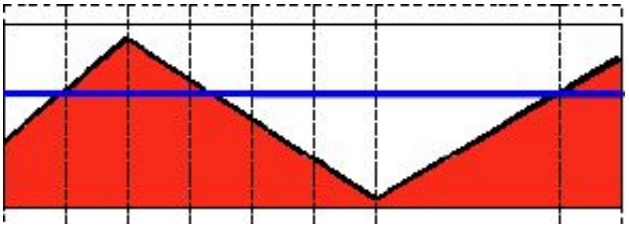
Appendix 1: Thresholding

Threshold is the simplest image segmentation method. It can be used to separate out regions of an image corresponding to objects or parts of the image which we want to analyze. This separation is based on the variation of intensity between the object pixels and the background pixels.

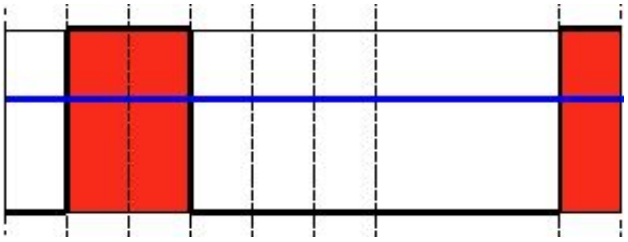
We do this by comparing the intensity of every pixel in this image with a value called the threshold, which is determined according to the problem at hand. The type of thresholding we use in this project is called binary thresholding. The function can be expressed as follows:

For illustrating the action of this function, if we take an image which has the following pixel intensity values:

$$\texttt{dst}(x,y) = \begin{cases} \texttt{maxVal} & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ 0 & \text{otherwise} \end{cases}$$



To this, if we apply the binary thresholding function, with the blue line denoting the threshold, we would get the following intensity plot:



There are also other types of fixed thresholding such Inverted Binary Thresholding, Truncation Thresholding, Thresholding to zero etc. Many libraries in MATLAB and Python provide functions to perform such thresholding operations.

Appendix 2: Quadratic Regression

The satellite map obtained from IMD website is an image in which each coordinate of a city corresponds to a specific pixel location. To map the coordinates to their corresponding pixels, we use a method called quadratic regression.

Quadratic regression helps provide a relationship between two data points. We are using quadratic regression instead of linear regression due to the curvature of the earth. The latitude is not linear and hence linear regression is not suitable for this case. We'd be able to obtain a quadratic function whose input is the coordinates and it would generate the pixel location corresponding to that coordinate.

The satellite image has a grid which numbers the coordinates on the map. We use each grid intersection and its pixel value to find its best fit quadratic function.

- Latitude

We note each grid intersection and find its pixel value in the y axis, and apply quadratic regression to it.

The parameters a,b and c are obtained and the accuracy is obtained from R^2. Value of R^2 as 1 means the equation is perfect with the data points.
After applying the quadratic regression we obtain the following parameters:

$$y_{pxl} \sim ax_{lat}^2 + bx_{lat} + c$$

STATISTICS

$R^2 = 0.9999$

RESIDUALS

e_1 plot

PARAMETERS

$a = -0.0728571$ $b = -18.9857$

$c = 1112.03$

x_{lat}	y_{pxl}
0	1114
10	911
20	703
30	481
40	234

Table A2.1 Grid intersections and pixel values

This is a really close approximation from the data points and hence is used to find out the vertical pixel value from latitude.

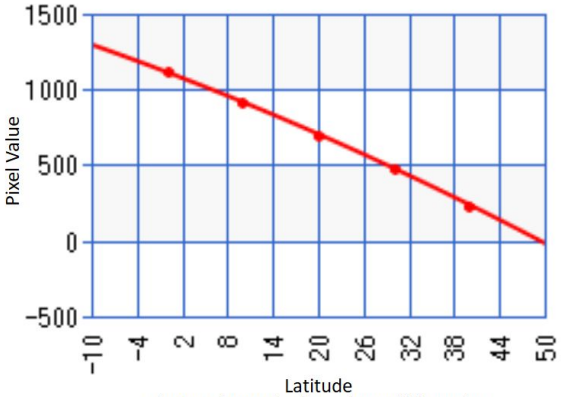


Fig. A2.1 The equation line and our grid data points

- Longitude

The horizontal grid lines are measured and tabulated, and similar computation is done. We observe the R2 value is exactly one which hints at a perfect correlation between longitude and horizontal pixel values. If we observe the difference between each pixel values, it is constant for longitude values but it's not constant for latitude. Hence quadratic regression is more suitable for latitude mapping.

$$y_{pxl} \sim ax_{lon}^2 + bx_{lon} + c$$

STATISTICS

$R^2 = 1$

RESIDUALS

e_1 plot

PARAMETERS

$a = 0.000178571$ $b = 20.1104$

$c = -894$

x_{lon}	y_{pxl}
50	112
60	313
70	515
80	716
90	917
100	1119

Table. A2.2 Longitudes and horizontal pixel values

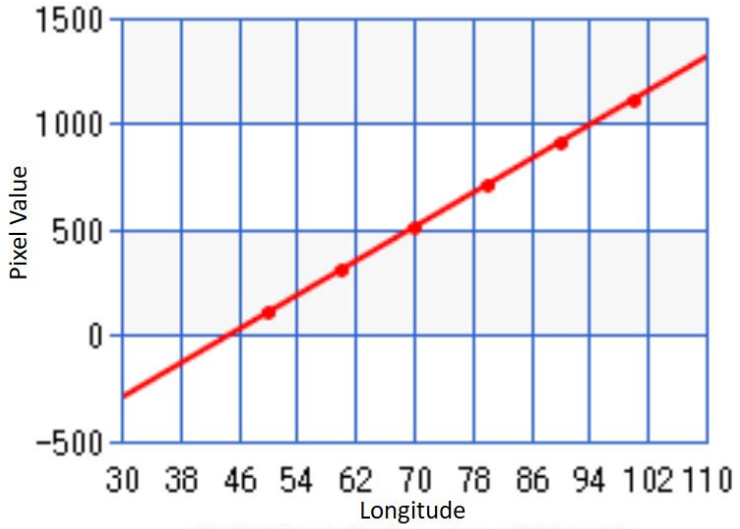


Fig. A2.2 The equation line and our grid data points

Appendix 3: Color Models (HSV and RGB)

Color models are visualisations that depict the color spectrum as multidimensional models. These models can be used by computer programs to represent colors. There are many common color models, the most common being RGB.

- RGB: This is a color model with three dimensions or channels- red, green and blue. These 3 colours can be mixed in varying amounts to produce any colour. When defining colours in this model, one has to know the values of all the primary colours constituting that colour. This can be conveniently visualised as a cube.

  However, the RGB color model is not an especially intuitive model for creating colors in code because humans do not think about colors as mixes of red, green, and blue lights.

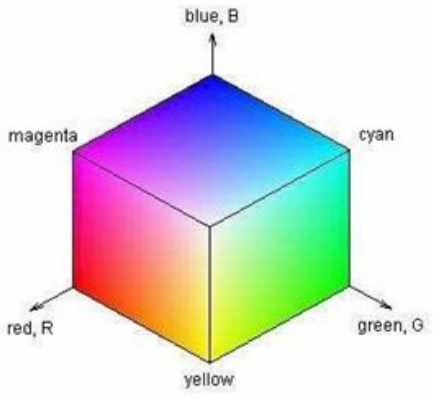


Fig.A2.1 Representation of RGB as a cube

- HSV: The HSV color space consists of 3 channels, where instead of the normal RGB channels we use Hue, Saturation and Value to determine the color of a pixel. It can be visualised through a cylinder.
  - The Hue roughly describes the type of color used, by an angle on the RGB color circle. 0 degrees hue results in Red, 120 in Green and 240 in Blue.
  - Saturation tells us the grayvalue. A 100% saturation will be the purest colour, while a 0% saturations gives grayscale
  - The Value tells us the brightness/ intensity of the color. A 0% value gives pure black, and 100% has no black mixed in it.

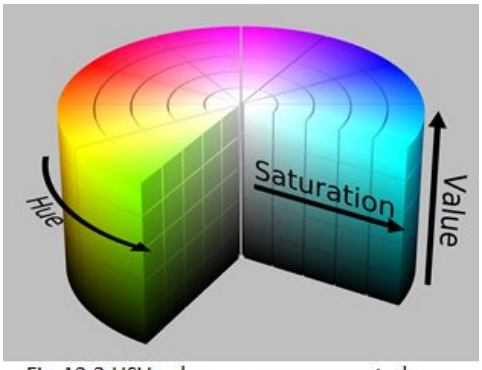


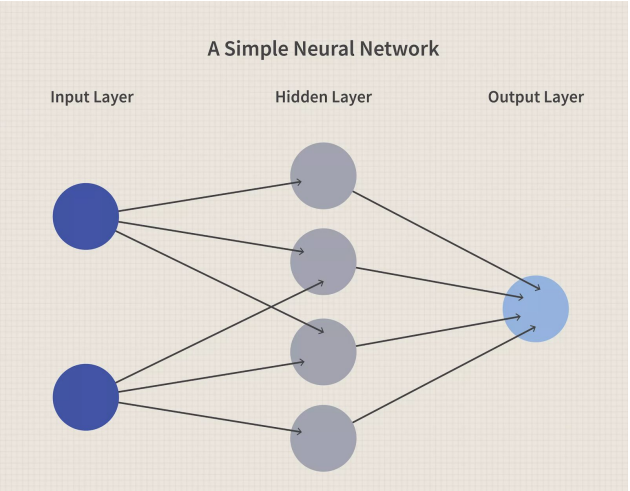
Fig.A2.2 HSV colour space represented as a cylinder

Appendix 4: Deep Learning

Deep learning is a subfield of artificial intelligence. Its main aim is to imitate the functioning of the human brain through algorithms designed to learn complex features from input data. It is a subset of the field of machine learning, and relies on an algorithm called Neural Networks.

A neural network is a learning algorithm that aims to recognise underlying patterns and features of data, in a way which mimics the functioning of the human brain. The human brain has several neurons, which is the basic working unit of the human brain. These are designed to transmit information to different parts of the bodies.

A neural network architecture tries to mimic this by connecting different computational units together, called nodes. These nodes are grouped into layers. The first layer is called the **input layer**, and the final layer is called the **output layer**. The intermediary layers are known as **hidden layers**. Different neural networks have different numbers of layers and nodes in these layers, giving rise to different architectures.



Each node applies a **non-linear activation function** to the input taken by it, which allows it to learn high-level features from the input data. Different matrices called "**weights**" are multiplied to the input data throughout the network, which determine the functions or features learned by the hidden layers. The hidden layers fine-tune these weights until the neural network produces low margins of errors on the training data. This margin of error is measured using a cost function.

A **cost function** is a function that measures the performance of a Neural network model on the training data. It calculates the error or deviation between the expected and predicted values, and presents it as a single number. Depending on the problem, there are different cost functions that can be used, such as: Mean absolute error, Mean squared error and Cross-entropy loss.

The cost function is reduced by using an **optimization algorithm**. Optimisation algorithms change the weights which are learned by the neural network, in order to minimize the cost function. There are different optimisation algorithms which are in use in the field of Deep learning, some of which are: Gradient Descent, Stochastic Gradient Descent and Adam.

So, the training of a neural network can be summarized as follows:

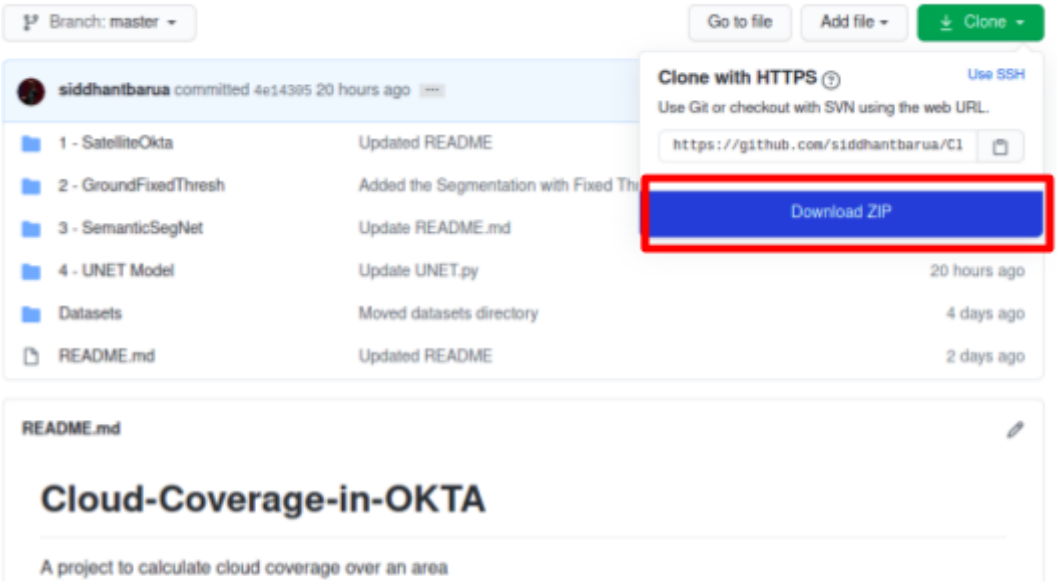
- The input data is fed through the neural network layers, to produce an output using the weights of the various hidden layers in the network.
- The predicted output is compared to the expected output, using a cost function to measure the deviations.
- An optimiser is used to minimize this error by changing the weight matrices.
- This process is repeated for several iterations. One iteration through the entire training dataset is known as one **epoch**.

After training the neural network for long enough for the cost function to converge to a minimum value, the neural network can be used to make predictions on other data which are not part of the test set.

Appendix 5: Installation of the Software

All of the project implementations, code and datasets are stored in our github repository: https://github.com/siddhantbarua/Cloud-Coverage-in-OKTA

The repository contains 4 different implementations as mentioned in the report. To install these on your device, visit the repository link and Clone the repository. This can be done by clicking on the “Clone” tab, and selecting “Download Zip” from the drop down menu. This is shown in the image below.



Cloning the project github repository

To run the first three implementations one needs to have MATLAB installed on their computer with a proper MATLAB licence. Directions for installing MATLAB can be found at this page: https://in.mathworks.com/help/compiler_sdk/dotnet/install-the-matlab-runtime.html

The 4th implementation requires python installed on your system.The guide for installing python can be found here: https://realpython.com/installing-python/

Once one has the adequate software installed, along with the project source code files, the various implementations can be used to calculate the cloud coverage in an image. The directions of use for every implementation are clearly stated in the README.md files in each of the directories.

For the 3rd and 4th implementations, the repository contains a directory named “Datasets/” which contains the SWINySEG dataset in zipped form. To train the network, one has to unzip the file, the password for which can be found at http://vintage.winklerbros.net/swinyseg.html. If one wants to only use the model for predictions using the pretrained weights, this step is not required.

REFERENCES

- Official IMD Website:
  https://mausam.imd.gov.in/

- About Okta units:
  https://www.metoffice.gov.uk/weather/guides/observations/how-we-measure-cloud

- Cloud Segmentation using thresholding:
  https://sites.wustl.edu/clouddetection/cloud-detection/fixed-and-adaptive-thresholding/

- Types of Thresholding:
  https://docs.opencv.org/2.4/doc/tutorials/imgproc/threshold/threshold.html

- Color Models and Color Spaces:
  https://programmingdesignsystems.com/color/color-models-and-color-spaces/index.html

- Semantic Segmentation: https://www.jeremyjordan.me/semantic-segmentation/

- SWIMSEG and SWINySEG datasets: http://vintage.winklerbros.net/resources.html

- About Convolutional Neural Networks:
  https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

  https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/

- About UNET: https://towardsdatascience.com/u-net-b229b32b4a71

GLOSSARY

- VIS - Image taken in Visible Spectrum
- IR Image - Image taken in the IR Spectrum which shows different temperatures in an image
- CNN - Convolutional Neural Network
- Filter - Kernels used in Convolution layer.
- Segmentation - Classifying pixels into different categories
- HSV - Color Space with 3 channels as Hue, Saturation and Value
- Artificial Neural Network -Artificial neural networks or connectionist systems are computing systems vaguely inspired by the biological neural networks that constitute animal brains.