Classification of cloud structures using semantic image segmentation

Group Project

ID1, ID2 | CM50265: Machine Learning 2 | 25.04.2021

ID	CONTRIBUTION
ID1	100%
ID2	100%

Trained model: Google drive link

Dataset: Kaggle link

Dataset: Google drive link

August 2020

List of Contents

List of Figures

List of Tables

1 Introduction

Convolutional neural networks are the state of the art to address the various computer vision tasks such as detection, localization, recognition, and image segmentation. Image segmentation is one of the central and most difficult practical problems of machine vision Davies (2012). This is due to the challenges involved in the process some of which are direction of lighting on the objects, viewpoint variation, scale variation, touching objects, computational intensity and most importantly lack of sufficient data for the use case being addressed. The most classic version of image segmentation is semantic segmentation Ghosh (2019). Image segmentation attempts to split the image into several regions each having a high level of uniformity in some parameter such as brightness, color, texture or even motion Davies (2012). Segmentation algorithms have been applied in several domains including medical image processing, pedestrian detection, defense guidance systems and forensics Ghosh (2019) . SegNet, UNet and Mask R-CNN are the state-of-the-art image segmentation architectures.

Due to the popularity of deep learning several image segmentation literature, libraries and methods have surfaced. Some of these have specialized in specific application areas whereas others have focused on the algorithms. Most of the online resources including Kaggle focus on the application of publicly available segmentation libraries. Knowledge of available methods would help to develop products but we wish to develop a better understanding of the underlying mechanics and methods to make us confident to tackle new challenges and develop better algorithms. Through our project we wish to answer the question of how an image segmentation algorithm is developed from scratch using a convolutional neural network, what kinds of preprocessing, data augmentation and post processing techniques are needed and evaluate our implementation by comparing it with a publicly available image segmentation library. To answer the question, we chose a problem from Kaggle involving image segmentation and found the problem of classifying different cloud structures from satellite images suitable to our question Ghosh (2019). Scientists at Max Planck Institute for Meteorology are researching the relationship between climates and cloud structures and have found that shallow clouds play a huge role in determining earth's climate *Understanding Clouds from Satellite Images* (2021). Through state-of-the-art image segmentation models scientists would be able to classify different types of cloud organization, thereby improving their physical understanding of these clouds, which in turn will help build better climate models. In this report the various image segmentation architectures are explored. In the methodology a design for the implementation of the segmentation model is finalized. Description of the coding done to implement the model is done in the implementation section. The results section discusses the results achieved with the model and compares its performance with other Kaggle contestants who use the library implementations. Finally, the conclusion section discusses conclusions drawn from our experiment as well as recommendations are made for future work in this area.

2 Background and Literature Review

We started our research by going through the different submissions made by contestants for the Kaggle competition for of cloud structure dataset at *Kaggle code* (2021). We observed that most contestants used the *Segmentation Models* (2021) library from Pavel Yakubovskiy. This library uses a transfer learning approach to implement the different segmentation models. Transfer learning uses feature extraction components of models pre-trained on large datasets such as such as VGG, Resnet, inception etc. which are then fed to segmentation model layers. The different segmentation models supported by the library were UNet, FPN, Linknet and PSPnet. Most contestants used UNet segmentation model from this library for the competition. This motivated us to investigate why UNet was popular.

UNet architecture consist of a contracting network and an expanding network structure Olaf, Philipp and Brox (2015). The contracting network comprises of convolutional layers coupled with down sampling layers to produce a low-resolution tensor containing the high-level information. The expanding network comprises of more convolutional layers coupled with up sampling layers to increase the size of the spatial tensor E, J and Fully (2017). The contracting network produces a tensor containing information about the objects and its features. The expanding network utilizes this information to produce segmentation masks. To obtain the spatial information, the high-resolution features from the contracting network are combined with the up sampled expanding network which helps the successive layers to assemble a more precise output. This combination of outputs from layers are called skip connections.

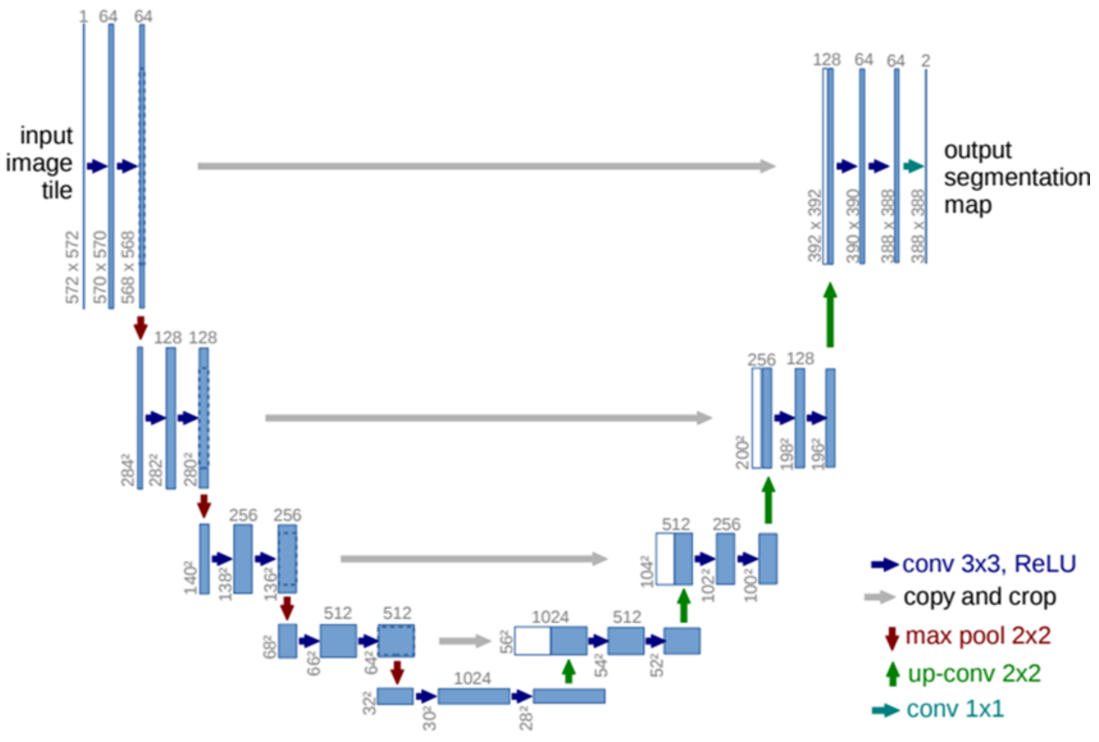


Figure 1: Reference UNet architecture from the paper.

We also researched the other state of the art Image segmentation architectures to decide on an architecture suitable for our project. Fully Convolutional Network (FCN) was initially developed as an extension to the classical CNN E, J and Fully (2017). It was developed to make classical CNN to take arbitrary size inputs by using only convolutional and pooling layers.

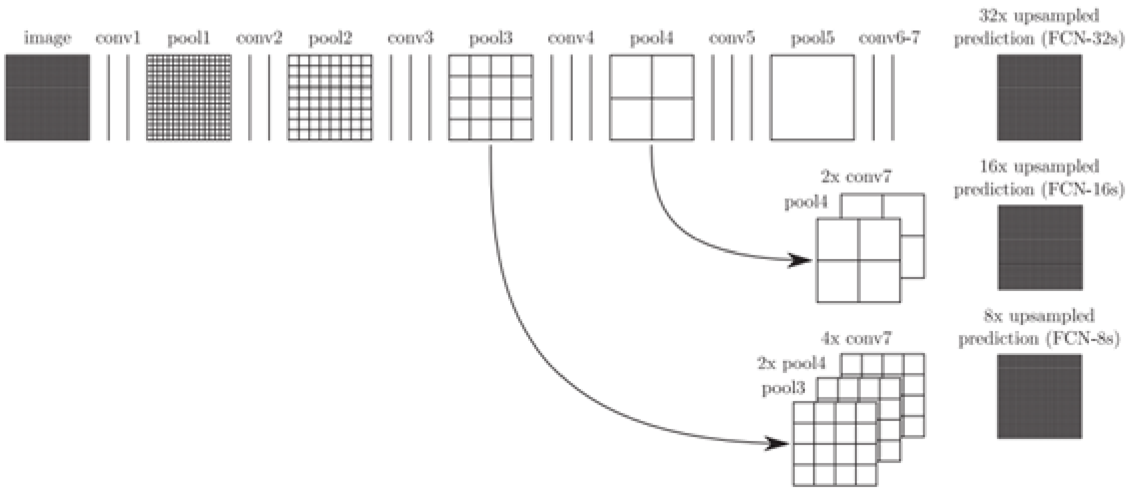


image
conv1
pool1
conv2
pool2
conv3
pool3
conv4
pool4
conv5
pool5
conv6-7
32x upsampled prediction (FCN-32s)
2x conv7
pool4
16x upsampled prediction (FCN-16s)
4x conv7
2x pool4
pool3
8x upsampled prediction (FCN-8s)

Figure 2: Reference FCN architecture from the paper.

As shown in figure below the FCN architecture uses various blocks of convolution and max pool layers to compress an image to 1/32 of its original size and predict the class of the image. Finally, a 32x up sampling is done to get the image back to its original size. The loss of spatial information when going deeper causes the direct predictions from this FCN also known as FCN-32 to be low in resolution. The shallow layers contain the location information. So, fusing them with the output helps to enhance the result. In FCN-16 the output from conv7 layer is 2x up sampled and fused (elementwise addition) with pool4 layer output and 16x up sampled. Similar operations are done for FCN-8 to improve results. FCN-8 has the best result.

The main difference between FCN and UNet are:

- FCN up samples only once. UNet is symmetric, there are same number of expansion and contraction layers.
- FCN uses single skip connections and fusing to increase accuracy, while UNet uses skip connections and concatenations.
- FCN uses interpolation in upscaling while UNet uses learnable weight filters

Mask RCNN is the state of the art for instance segmentation Kaiming et al. (2020). While semantic segmentation classifies all the pixels that belong to a class, instance segmentation goes deeper and classifies instances of each class in addition to identifying the class of the pixel. Compared to UNet and FCN it is more complex due to its different branches: a branch for predicting segmentation masks on each region of interest in parallel with a branch for classification and bounding box regression Scherr et al. (2018). Due to time limitations for the project, we limited our focus to semantic segmentation models.

3 Methodology

As we wanted to learn how to implement semantic segmentation algorithm from scratch, we found UNet model easy to implement, trainable from scratch, and expandable, e.g., using stacking and residual connections. Moreover, UNet architecture by design consists of an end-to-end image segmentation solution and it is widely used in medical image segmentation. UNet also combines multi-scale features, and this is necessary for accurate segmentation E, J and Fully (2017). UNet is known to yield precise segmentation with fewer training images E, J and Fully (2017). As with most machine learning projects, we divided our model development into different stages:

1. Exploratory data analysis: Analyze the distribution of cloud types between images. This helps to evaluate is we have imbalanced data.
2. Data preprocessing: Analyze the ground truth data arrangements and re arrange it if necessary, it to a compatible format for the model to output.
3. Do train test splits: Split the train dataset into a train dataset and validation dataset. The validation dataset will be used for hyperparameter tuning as well as passed to the model to evaluate the loss and any model metrics at the end of each epoch.
4. Create data generators for the model: We will use our own custom data generators to feed data to the model as we may need to perform image operations and data augmentations to the data.
5. Create the UNet model: Build model using the chosen neural network library API's
6. Train the model: Train model with some default parameters to validate functionality before hyperparameter tuning.
7. Hyperparameter tuning: Optimize the model by finding best hyperparmaters and retrain the model with those parameters.
8. Do post processing: For better visualization we apply thresholds to pixel values.
9. Visualize predictions on validation data: Demonstrate model accuracy.
10. Compare performance with library implementation: Train similar model from library to evaluate our model's performance.

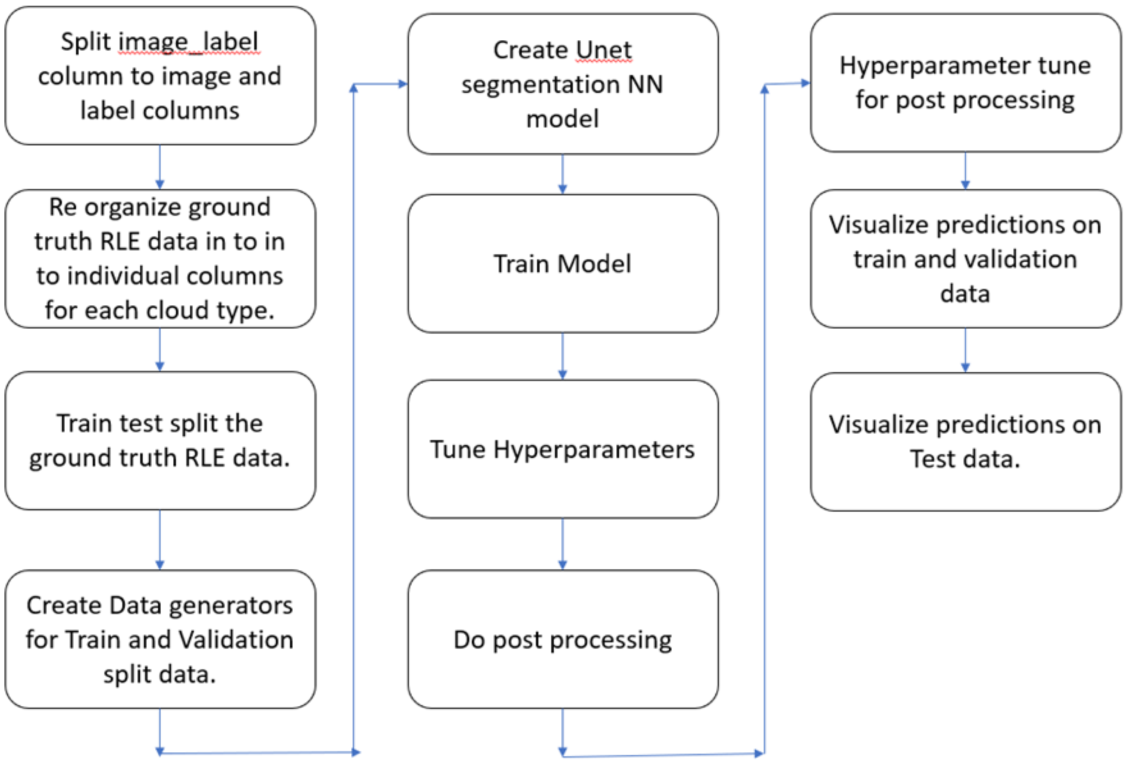

Split image_label column to image and label columns
Re organize ground truth RLE data in to in to individual columns for each cloud type.
Train test split the ground truth RLE data.
Create Data generators for Train and Validation split data.
Create Unet segmentation NN model
Train Model
Tune Hyperparameters
Do post processing
Hyperparameter tune for post processing
Visualize predictions on train and validation data
Visualize predictions on Test data.

Figure 3: Stages of model development

4 Implementation

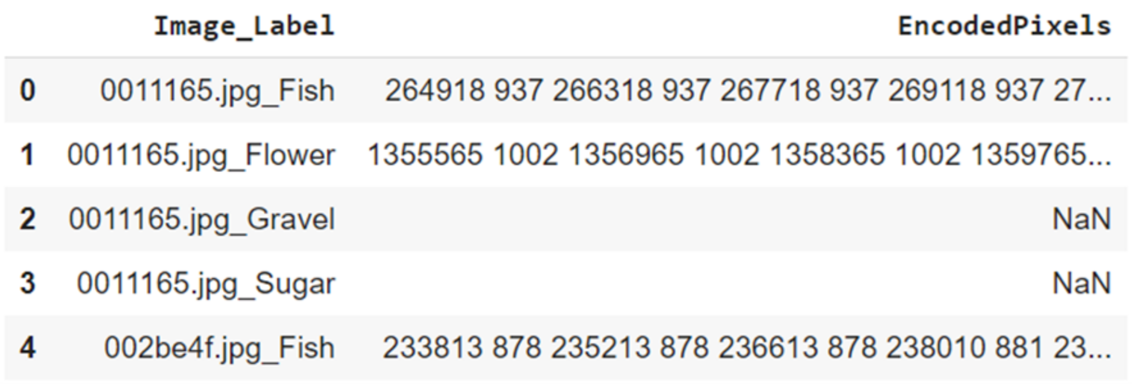
4.1 Environment

As one of the project requirements was to submit a Google colab notebook, the model development was done in Google Collaboratory. Google Colab or Collaboratory is a free cloud platform Machine Learning supported by Google. It allows its users to use free CPU Tesla GPU and TPU services. As a normal user the duration of GPU runtime session was limited. So, an upgrade to Colab Pro was made for a monthly fee. Even with Colab Pro account it was noticed that there was a hard cut out of the GPU session after 12 hours. So hyperparameter tuning part of the project was moved to the NITT server of the Hex GPU cloud. NITT server offers the 4 GPU's to use which are GeForce RTX 3090's.

4.2 Data preprocessing

The Kaggle dataset consists of test and train directories of 2100x1400 resolution jpeg satellite images of cloud structures. Ground truth data is provided in a train.csv' excel sheet. Each image in the 'train_images' directory is represented by 4 rows in the 'train.csv' excel sheet. The name of the image is suffixed separately with each of the cloud label we are attempting to classify. If a particular cloud type is present in an image, the row corresponding to the image name followed by the cloud type will have an entry. In order to save space pixel values of each cloud type are run length encoded (RLE).

Pixels are represented as pairs of values consisting of a start value and end value. For example, a pair 1,3 would represent pixels 1,2 and 3. Importantly the pixels are numbered from top to bottom then left to right. So, considering pixels to be arranged in an array pixel 1 would have indices 1,1



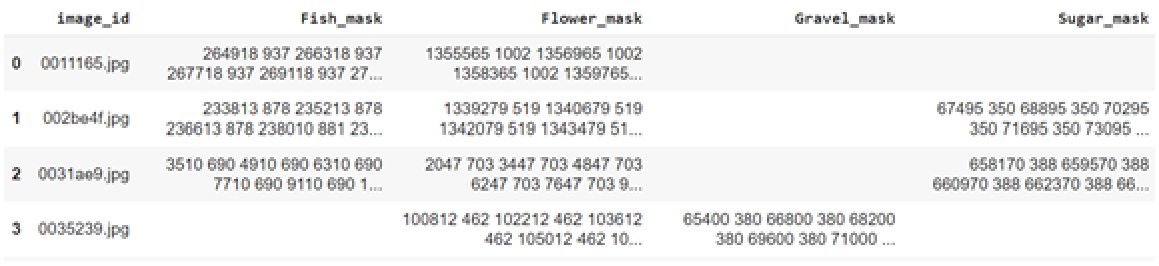
	Image_Label	EncodedPixels
0	0011165.jpg_Fish	264918 937 266318 937 267718 937 269118 937 27...
1	0011165.jpg_Flower	1355565 1002 1356965 1002 1358365 1002 1359765...
2	0011165.jpg_Gravel	NaN
3	0011165.jpg_Sugar	NaN
4	002be4f.jpg_Fish	233813 878 235213 878 236613 878 238010 881 23...

Figure 4: Ground truth representation

and pixel 2 would have indices 2,1.

The input for the model were chosen to be the images. Image size was chosen to be 384x384 to be divisible by 32 and to save on the training time. The standard input sizes are between 200x200 and 600x600 *A Beginner's guide to Deep Learning based Semantic Segmentation using Keras* (2021). Resizing of the input images were done in the respective Data generators for the neural network.

Outputs of the models were chosen to be 4 different masks for each class of cloud type. For this reason, the ground truth data in 'train.csv' was first reorganized into a python Pandas frame as in following figure.



	image_id	Fish_mask	Flower_mask	Gravel_mask	Sugar_mask
0	0011165.jpg	264918 937 266318 937 267718 937 269118 937 27...	1355565 1002 1356965 1002 1358365 1002 1359765...		
1	002be4f.jpg	233813 878 235213 878 236613 878 238010 881 23...	1339279 519 1340679 519 1342079 519 1343479 51...		67495 350 68895 350 70295 350 71695 350 73095 ...
2	0031ae9.jpg	3510 690 4910 690 6310 690 7710 690 9110 690 1...	2047 703 3447 703 4847 703 6247 703 7647 703 9...		658170 388 659570 388 660970 388 662370 388 66...
3	0035239.jpg		100812 462 102212 462 103612 462 105012 462 10...	65400 380 66800 380 68200 380 69600 380 71000 ...	

Figure 5: Ground truth re-arranged

The RLE mask corresponding to each cloud type were converted in to mask images and stored in to a 384x384x4 numpy array and this became the ground truth for the image. The masks were also resized to 384x384 in the Data generators. This enables the output of the segmentation model to be a 384x384x4 numpy array for an image input where each of the 4 for indexes of the array represent the predicted mask for each of the cloud type in the order shown in figure above.

4.3 Performance Metric

Mean dice coefficient was chosen as the performance metric as it is widely used in validation of image segmentation algorithms Scherr et al. (2018) and is equivalent to F1 score *F1 Score = Dice Coefficient* (2021). Also Dice coefficient is known to work with imbalanced data. Dice coefficient is used to calculate the pixelwise agreement between and predicted segmentation and its ground truth *Kaggle Evaluation* (2021). Dice coefficient is calculated as in following figure. where X is the

$$\frac{2 * |X \cap Y|}{|X| + |Y|}$$

Figure 6: Dice coefficient calculation

predicted set of pixels and Y is the ground truth.

4.4 Loss function

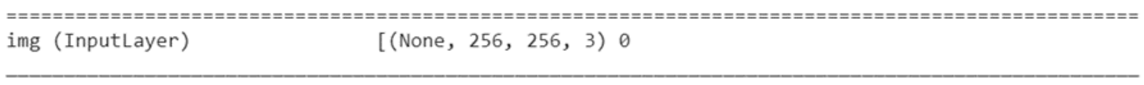
The ground truth data was rearranged in such a way that each image input produces a single 384x384x4 numpy array as output. Each of the 4 indexes represent a flower, fish , gravel or sugar mask. Since masks are either 0 or 1 pixels, they represent the labels for our problem. Binary Cross Entropy (BCE) loss was found to be a suitable loss function for our problem Scherr et al. (2018). BCE compares the probability of each prediction with the actual class output which may be either 0 or 1. It then calculates a score that penalizes the probabilities based on the distance from the expected value. Most Kaggle participants were found using a combination of BCE and Dice loss. Dice loss originates from the Dice coefficient *Kaggle code* (2021). It is used to overcome the limited scope of cross entropy loss in predicting boundaries. The following paper Scherr et al. (2018) too used this combination loss function and noted that occasionally when there is only one object this combination loss function provides some improvement. Dice loss is calculated by doing 1 – Dice coefficient *Understanding Dice Loss for Crisp Boundary Detection* (2021). BCE gives the binary classification of each pixel when compared to the ground truth mask (Mask generated from RLE are either 0 or 1's). BCE treats every pixel as an independent prediction and dice loss looks at the resulting mask in a holistic manner. We decided to use the combination loss function and experiment the impact of the use of loss functions individually. In addition we also tried using Jaccard distance which measures the similarity between two datasets. It is defined as the size of the intersection divided by the size of the union of the sample sets. It is mostly used in binary data. Using Jaccard distance gave a comparatively lower Validation dice coefficient. Using the combination of BCE dice coefficient and Dice loss function gave us the highest validation dice coefficient value.

4.5 Library to build model

We found the syntax of Keras to be slightly simple and many public tutorials were available to learn the apis quickly in addition to Keras's own tutorials. We could not find the same number of tutorials for Pytorch in the area of image segmentation. When trying out different image segmentation tutorials in Google colab we found Pytorch was giving compatibility errors with certain libraries due to incompatible TensorFlow versions. So, we considered Keras to be relatively stable compatibility wise at least in our problem space.

4.6 Model implementation

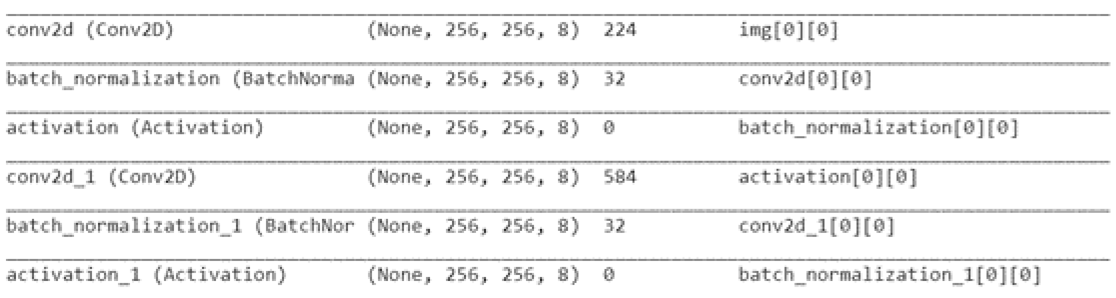
Input image size was set at 384x384 to speed up training also it is known that bigger image sizes increase the model's complexity but gain in accuracy is not that significant. Experimenting with size 512x512 showed significant increase in training time both on Hex and Google olab and occasionally gave OOM errors on colab.



```
==========================================================================================
img (InputLayer)              [(None, 256, 256, 3) 0
__________________________________________________________________________________________
```

Figure 7: Input layer

After the input layer we implement the repeated convolution layer in the contracting network. This layer applies 3x3 unpadded convolutions followed by ReLU activation function. The number of filters remain the same when applying a repeated convolution layer but gets doubled every subsequent layer of the contracting network. We experimented with different filter numbers such as 8, 16 and 32 and found that the performance metric did not improve significantly but training time increased significantly. So, we chose 8 filters for the starting layer and doubled it for every repeated convolutional layer in the expanding network. Batch normalization and 'he_normal' kernel initializer were used. While experimenting without Batch normalization it was found that the performance metric did not increase and was taking longer time to train. This was indicative of chasing a moving target in the layers, which using batch normalization help to remove. As we did not have time to experiment if batch normalization would be good before or after the activation function, we decided to place the batch normalization function before the activation function.

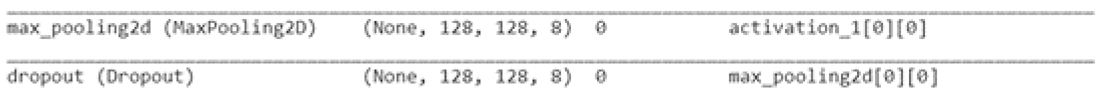


```
__________________________________________________________________________________________
conv2d (Conv2D)                (None, 256, 256, 8)  224         img[0][0]
__________________________________________________________________________________________
batch_normalization (BatchNorma (None, 256, 256, 8)  32          conv2d[0][0]
__________________________________________________________________________________________
activation (Activation)        (None, 256, 256, 8)  0           batch_normalization[0][0]
__________________________________________________________________________________________
conv2d_1 (Conv2D)              (None, 256, 256, 8)  584         activation[0][0]
__________________________________________________________________________________________
batch_normalization_1 (BatchNor (None, 256, 256, 8)  32          conv2d_1[0][0]
__________________________________________________________________________________________
activation_1 (Activation)      (None, 256, 256, 8)  0           batch_normalization_1[0][0]
```

Figure 8: Repeated convolution layer

Alexnet initialized weights for the layers by drawing from a zero-mean Gaussian distribution with standard deviation 0.01. This helped them to accelerated early stages of learning by providing ReLUs with positive inputs Krizhevsky, Sutskever and Hinton (2017). Similar technique is applied by using 'He_normal' kernel initializer for our network.

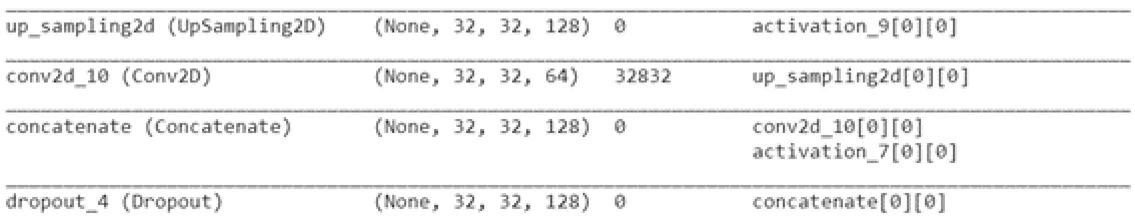
The Max pooling layer with stride 2 causes down sampling and so it is called the contracting network. Since we added batch normalization, we also added a dropout layer at the end of each repeated convolution block to prevent over-fitting. The repeated convolution layer described previously is applied four times in the contracting network and each down sampling step doubles the number of feature channels.



```
__________________________________________________________________________________________
max_pooling2d (MaxPooling2D)   (None, 128, 128, 8)  0           activation_1[0][0]
__________________________________________________________________________________________
dropout (Dropout)              (None, 128, 128, 8)  0           max_pooling2d[0][0]
```

Figure 9: Max pooling and dropout layers

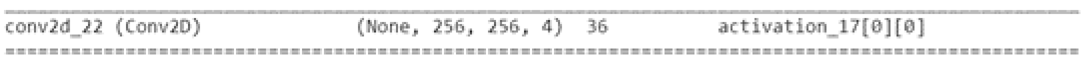
The expanding network which is symmetric to the contracting network implements two main functionalities. The pooling layer of the contracting network is replaced with an up-sampling layer which causes an increase in resolution of the output and a 2x2 convolution which halves the number of feature channels. The output of the corresponding layer in the contracting network is then concatenated with the output of this layer. Dropout is applied and the replicated convolutions is performed on the output of the dropout layer.



```
up_sampling2d (UpSampling2D)    (None, 32, 32, 128)  0        activation_9[0][0]
conv2d_10 (Conv2D)              (None, 32, 32, 64)   32832    up_sampling2d[0][0]
concatenate (Concatenate)       (None, 32, 32, 128)  0        conv2d_10[0][0]
                                                              activation_7[0][0]
dropout_4 (Dropout)             (None, 32, 32, 128)  0        concatenate[0][0]
```

Figure 10: Up sampling and concatenate layers in expanding network

The output layer consists of a convolutional layer with 4 filers and a 1x1 kernel to map the output to a 384x384x4 array which consists of 4 masks for each cloud type.



```
conv2d_22 (Conv2D)              (None, 256, 256, 4)  36       activation_17[0][0]
==================================================================================================
```

Figure 11: Output layer

4.7 Custom data generator implementation

Since the images and ground truth data needed preprocessing before feeding to the UNet we implemented our own data generators for the train and validation data splits. In the train data generator, we resized the input images to 384x384 resolution. We then applied samplewise_std_normalization transform to standardize the images. The RLE data corresponding to each cloud type was first converted to mask, resized to 384x384 and then appended to a numpy matrix of size 384x384x4. Similar data generator was created for validation. Having two different data generators was a provision to add data augmentations if needed to the train data.

4.8 Hyperparameter Tuning

Weights and Biases (Wandb) *Website* (2021) , provider of a platform for enabling collaboration and governance across teams building machine learning models. Weights & Biases Sweeps were used to automate hyperparameter optimization. Wandb provides live metrics, terminal logs and system statistics streamed to a centralized online dashboard which can be accessed from anywhere. To use Wandb Sweeps for hyperparameter tuning a couple of lines of code were added mainly to login to the Wandb account , create a Wandb config containing values to be tried, initialize a sweep and create agents that would run on the machine to train the models with the values from the config *Hyperparameter Tuning for Keras and Pytorch models* (2021).

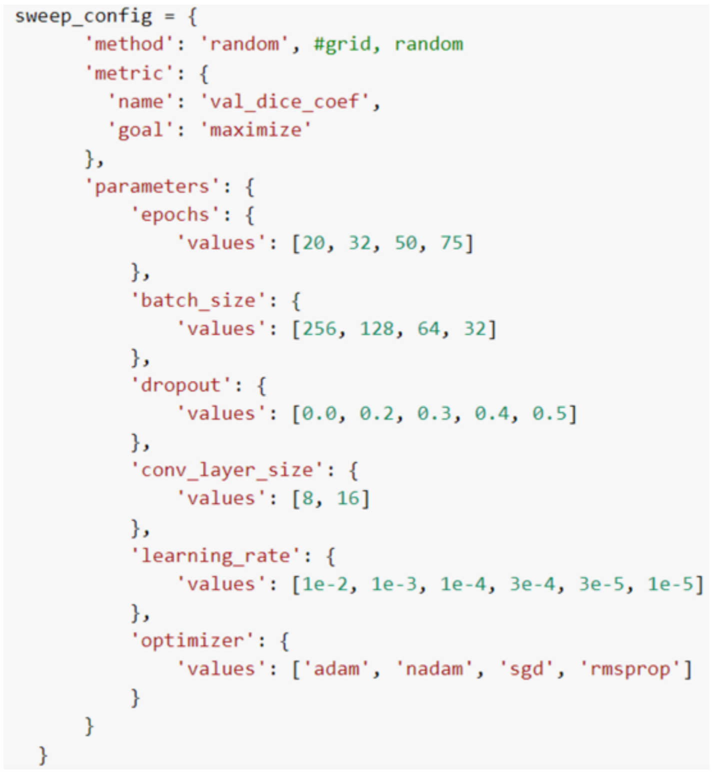

```
sweep_config = {
      'method': 'random', #grid, random
      'metric': {
        'name': 'val_dice_coef',
        'goal': 'maximize'
      },
      'parameters': {
          'epochs': {
              'values': [20, 32, 50, 75]
          },
          'batch_size': {
              'values': [256, 128, 64, 32]
          },
          'dropout': {
              'values': [0.0, 0.2, 0.3, 0.4, 0.5]
          },
          'conv_layer_size': {
              'values': [8, 16]
          },
          'learning_rate': {
              'values': [1e-2, 1e-3, 1e-4, 3e-4, 3e-5, 1e-5]
          },
          'optimizer': {
              'values': ['adam', 'nadam', 'sgd', 'rmsprop']
          }
      }
  }
```

Figure 12: Hyperparameters to tune in wandb config

Initially the hyperparameter tuning was started on Google colab and it ran for seven sweeps but was interrupted due to time out after 12 hours. So hyperparameter tuning was moved to Hex cloud. A separate python script was created for hyperparameter tuning alone to enable the hyperparameter tuning the progress in a Linux screen session. The hyperparameter parameter script also enabled commandline arguments which allowed experimentation with different image sizes and filter sizes in multiple Linux screen sessions. The script can be found in the Appendix. Even in the cloud, sweeps were interrupted due to random network and server issues. So we have a set of different sweeps over different set of parameters. The Wandb website produces interactive graphs showing impact of the different hyperparameters on the metric which in our case was the dice coefficient on the validation data called val_dice_coeff. All the graphs are visible from the wandb dashboard for our project *Dashboard* (2021)

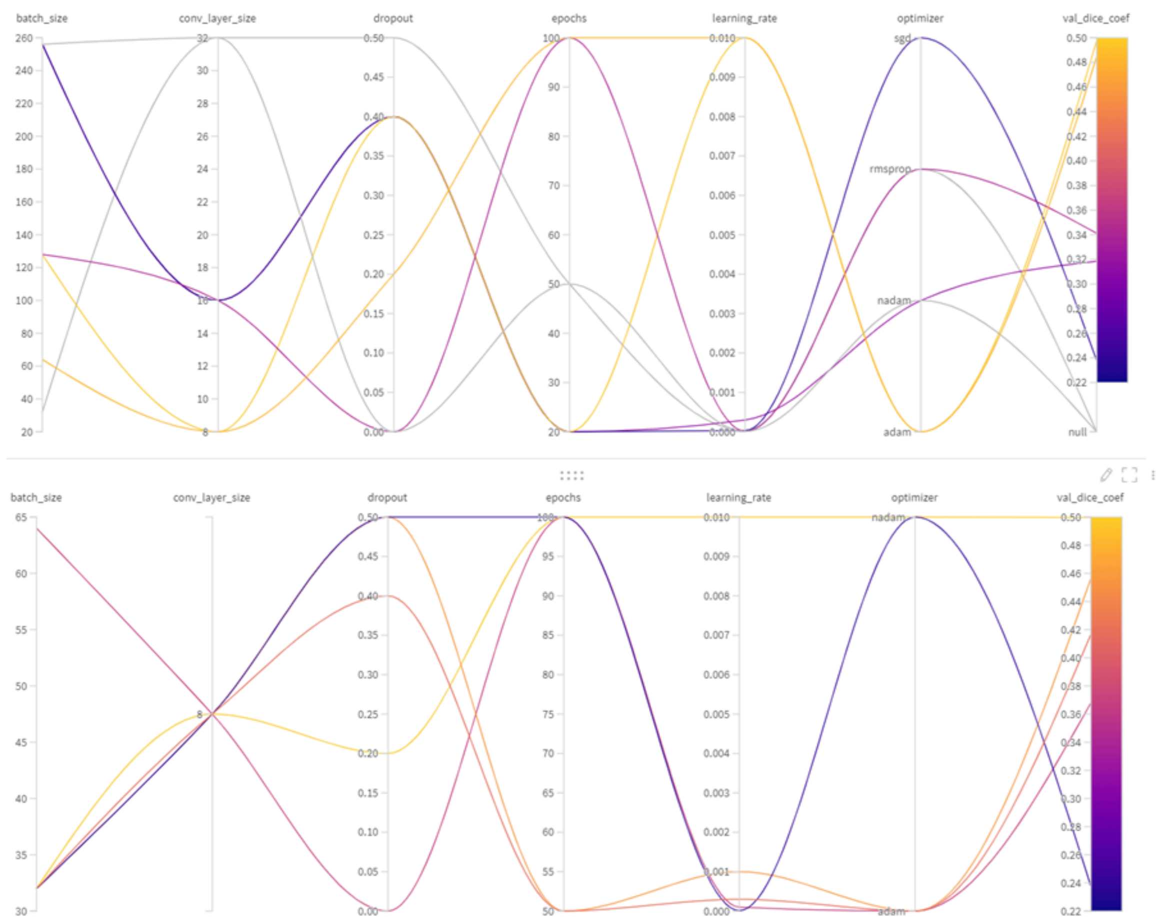


Figure 13: Wandb sweeps after 7 and 8 runs

After 10 runs the optimal hyper parameters were found to be:

Batch size	32
Epochs	50
Dropout	0.2
Optimizer	Adam
Batch-Normalization	True
Filter size	8

Figure 14: Optimal hyperparameters after tuning

The model was then retrained with the hyperparamters and the following learning and loss curves were obtained.

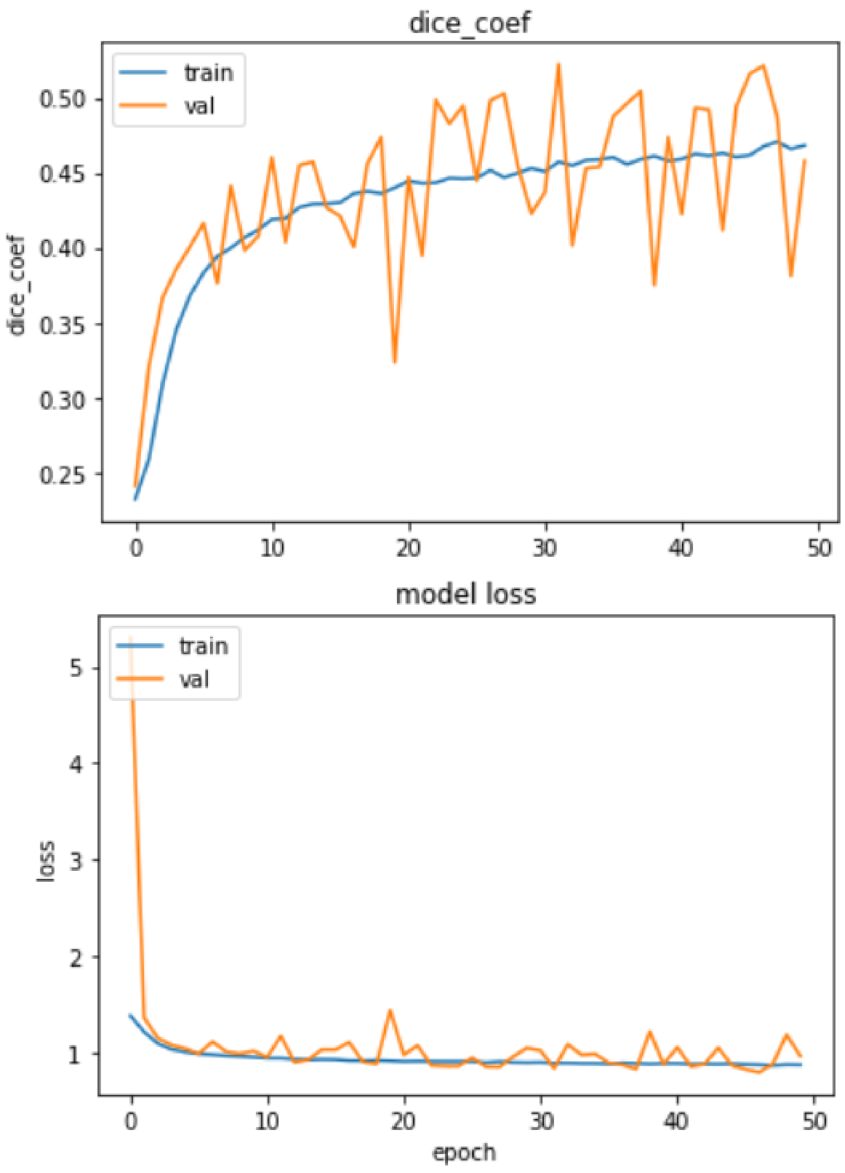
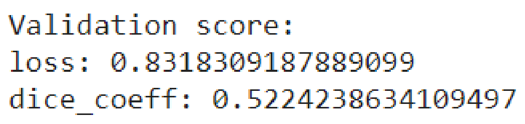


Figure 15: Curves after re training with hyperparameters

5 Evaluation

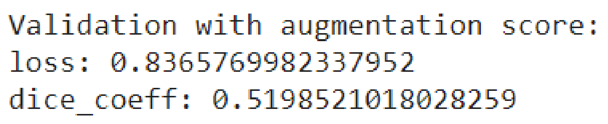
Evaluation of the model was done in two stages using the Keras model evaluate API. Evaluate method returns the loss value and metrics values for the model in test mode. Evaluation is done in batches. In the first stage we called the evaluate method on the same validation data that was used during model training. We receive the same results as the training phase which was expected.



```
Validation score:
loss: 0.8318309187889099
dice_coeff: 0.5224238634109497
```

Figure 16: Evaluation result without validation augmentation

In the second stage we call evaluate method with augmented validation data. We use image augmentation library which is described in 6.2. To demonstrate efficiency of our model in predicting augmented validation data which will represent unseen data for our project. Our custom validation loader provides option to pass an augmentation pipeline structure. Augmentation pipeline is a set if image transformations that will be applied when augmentation is requested. Using augmented validation data we get the following results.



```
Validation with augmentation score:
loss: 0.8365769982337952
dice_coeff: 0.5198521018028259
```

Figure 17: Evaluation result with validation augmentation

From the above results we see that the validation dice coefficient is approximately equal to the validation dice coefficient obtained during training.If our model was over-fitting we would have got a lower validation dice coefficient. Over-fitting occurs when the model tries to learn the noise and inaccurate values in the data set thereby causing a loss in efficiency and accuracy of the model. We had included drop out and batch normalization while building our model to specifically address this issue. Based on this result we can conclude our model does not over fit and generalises well.

6 Results

The top scores for the Kaggle competition were in the range from 0.60 to 0.67 *Kaggle Leaderboard* (2021) . These scores were calculated from the model's predictions on the test dataset for which we do not have the ground truth information. So, the alternative we chose was to compare the mean dice coefficient on the validation dataset of top implementations. Validation data split was done before training and made sure no data leakage happens. Our validation mean dice coefficient was improved in following stages:

6.1 Validation mean dice coefficient after hyperparameter tuning

After retraining the segmentation model using optimal hyperparameters as shown in previous section a validation mean dice coefficient value of 0.50 was obtained. This is similar to what was visible from the Wandb sweep graph.

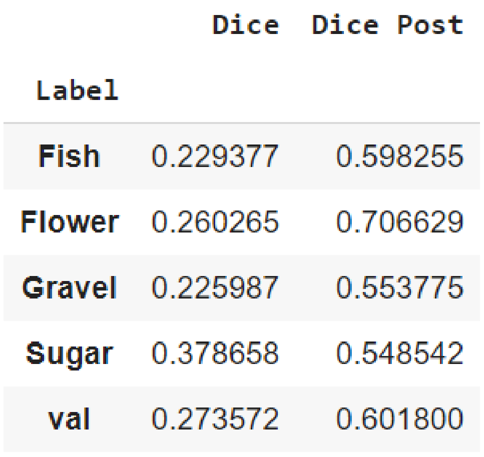
6.2 Validation mean dice coefficient after image augmentation

To improve the validation mean dice coefficient, we first applied image augmentation. Image augmentation is used to generate synthetic data from existing training samples by augmenting the samples via number of random transformations. Though the Keras default data generator had options to apply transforms on the images we could not use it as we needed to apply agumentations to both the image as well as its corresponding mask. This is a feature supported by the image augmentation library called Albumentations *Albumentations* (2021). This library contains more than 70 different augmentations and works with both Keras and pytorch libraries. We used the library API to experiments with a range of image augmentations such as Horizontal flip with 50% probability, vertical flip with 50% probability and shift scale rotate with 50% probability. By using probabilities for applying transforms, the data gets transformed every epoch and this helps improve quality of the model. After applying image augmentation, the validation dice coefficient increased 0.52

6.3 Validation mean dice coefficient after post processing

The following paper Scherr et al. (2018) suggests using of simple thresholding post-processing to improve model performance. Thresholding is the most basic form of segmenting images which replaces a pixel whose value is less than a pre-defined threshold with zero and replace a pixel whose value is more than the threshold with 1. We use OpenCV cv2.threshold function to do the thresholding with a provided threshold value.

In addition to thresholding, we also separate cloud from the background through connected component labelling using cv2.connectedComponents API. Connected component labelling *Connected Component Labeling* (2021) solves the problem of finding out parts of an image that are connected physically irrespective of color. Connected components in an image are the set of pixels with same value connected through a connectivity rule. There are two types of connectivity rules "8-way" or '4-way'. The former connects all pixels of the same value along the edges while the latter connects all pixels with same values along the corners. The default connectivity is '8-way'. The cv2.connectedComponents API returns N number of labels [0, N-1] where 0 represents the background label. We then decide which labels are set to 1 by providing a threshold mask size value. Both mask size and pixel threshold values are hyperparameters which are tuned to get best validation mean dice coefficient on the validation dataset. Since each cloud type has its own set of mask size and threshold value random search hyperparameter tuning is done.



Label	Dice	Dice Post
Fish	0.229377	0.598255
Flower	0.260265	0.706629
Gravel	0.225987	0.553775
Sugar	0.378658	0.548542
val	0.273572	0.601800

Figure 18: Validation mean dice score with post-processing

With post processing we obtained a final validation mean dice score of ***0.60***

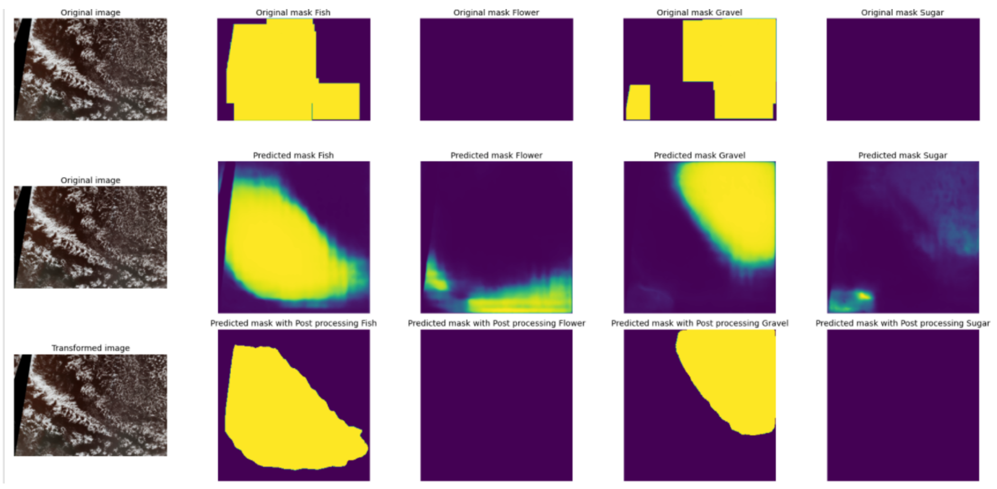
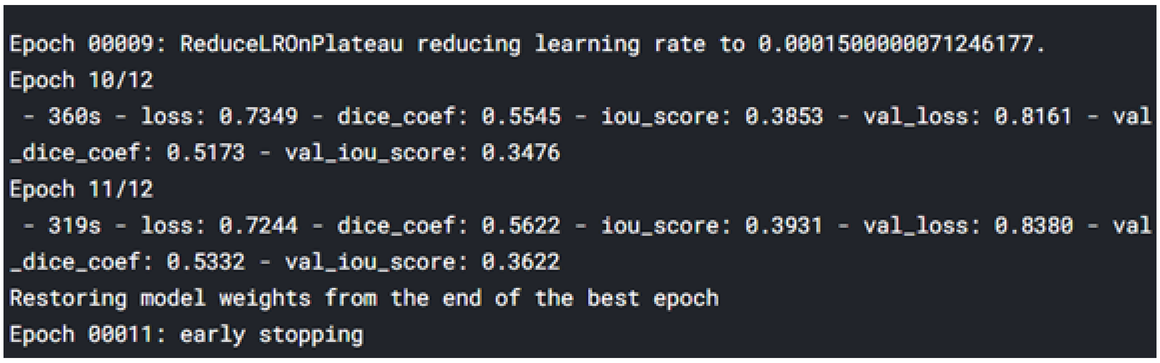


Figure 19: Model output Visualization with and without post-processing

6.4 Comparison with alternative method

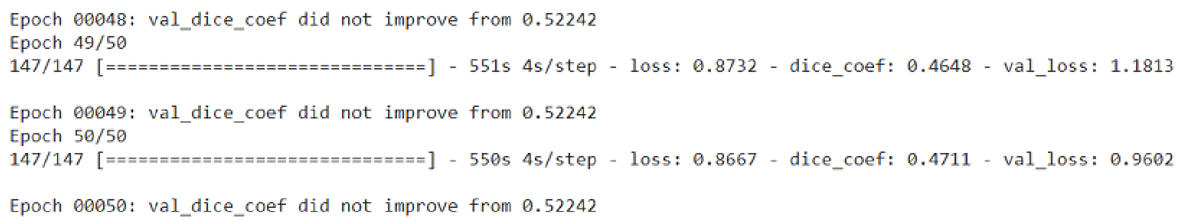
As discussed in background and literature review section we noticed most contestants used the 'segmentation_models' (SM) library from Pavel Yakubovskiy for their submissions. The UNet implementation from this library along with the Resnet34 backbone was the common choice for most contestants. Resnet34 is a 34 layer convolutional neural network that is a state-of-the-art image classification model. This is a model was pre-trained on the ImageNet dataset–a dataset that has 100,000+ images across 200 different classes. It is different from traditional neural networks in the sense that it takes residuals from each layer and uses them in the subsequent connected layers *Resnet34* (2021). We looked for a submission that earned gold in the competition and had used the SM library. The following submission *Kaggle submission* (2021) had used the SM library along with some Kaggle utility scripts available specific to the problem. They had obtained a validation mean dice score of 0.53



```
Epoch 00009: ReduceLROnPlateau reducing learning rate to 0.00015000000071246177.
Epoch 10/12
 - 360s - loss: 0.7349 - dice_coef: 0.5545 - iou_score: 0.3853 - val_loss: 0.8161 - val
_dice_coef: 0.5173 - val_iou_score: 0.3476
Epoch 11/12
 - 319s - loss: 0.7244 - dice_coef: 0.5622 - iou_score: 0.3931 - val_loss: 0.8380 - val
_dice_coef: 0.5332 - val_iou_score: 0.3622
Restoring model weights from the end of the best epoch
Epoch 00011: early stopping
```

Figure 20: Competition Validation mean Dice coefficient

We then replicated the code which used library to create the same model in our environment. We trained the model using our data generators and compared the mean validation coefficient values. The validation mean dice coefficient using the library was 0.53 after which the model started to over-fit. This confirms that the library worked identically with the competition.



```
Epoch 00048: val_dice_coef did not improve from 0.52242
Epoch 49/50
147/147 [==============================] - 551s 4s/step - loss: 0.8732 - dice_coef: 0.4648 - val_loss: 1.1813

Epoch 00049: val_dice_coef did not improve from 0.52242
Epoch 50/50
147/147 [==============================] - 550s 4s/step - loss: 0.8667 - dice_coef: 0.4711 - val_loss: 0.9602

Epoch 00050: val_dice_coef did not improve from 0.52242
```

Figure 21: Our Validation mean Dice coefficient

Our validation mean validation dice coefficient was 0.52 even without post processing, which demonstrates that the performance our segmentation model is on par with library implementation of the segmentation model which used a transfer learning with Resnet34 backbone.

7 Conclusion and Future Work

The projects aimed to research into the wide field of image segmentation knowledge, Convolutional neural networks, their applications in image processing and machine learning algorithm development processes. We were able to incorporate our research in to designing a efficient image segmentation model

For the background study, a through literature survey was conducted in the area of semantic image segmentation techniques such as FCN, UNet and Mask RCNN. By weighing pros and cons of each of the algorithm and keeping project scope in mind UNet image segmentation algorithm was chosen to be implemented.

Following the typical machine learning project development process, the project was divided in to multiple stages starting from data preprocessing all the way to model output visualization. Development of stages was parallelized between team members to speed up development process. For example, comparison part, model output visualization part could be developed parallelly with the model development itself. Necessary testing and evaluation of each subsystem was performed to guarantee expected functionality output from each stage.

Scripting of the hyperparameter tuning code to use command line arguments helped to experiment with different parameter values individually on the Hex cloud without running full fledged hyperparameter tuning. Script also helped running parallel Linux screen sessions in the Hex cloud which saved time a lot. Comparing with segmentation library results and running the library itself locally helped to evaluate the efficiency of our model. We also experimented with post processing using thresholding and connected component labelling which helped improve the visualization of the results greatly by removing intermediate pixel values in the predicted masks.

UNet segmentation architecture indeed proved to be an efficient algorithm that could be implemented from scratch. Its performance could have been improved though experimentation with pre-trained backbones, stacking and residual connections given the time and resources. Through this project we expect to have met all requirements for the project and succeeded in answering the research question that we set out to answer.

For future work We would consider implementing Mask RCNN as it is the state of the art in instance segmentation and feature learning is configurable at individual layers. Nested UNet or UNet++ is another power segmentation architecture used in medical image segmentation. It is UNet architrcture with redesigned skip connections. The knowledge we gained through this project will help to understand implement these architectures effectively.

8 References

Albumentations, 2021. Available from: https://github.com/albumentations-team/albumentations [Accessed 2021-04-27].

A beginner's guide to deep learning based semantic segmentation using keras, 2021. Available from: https://divamgupta.com/image-segmentation/2019/06/06/deep-learning-semantic-segmentation-keras.html [Accessed 2021-04-27].

Connected component labeling, 2021. Available from: https://iq.opengenus.org/connected-component-labeling/ [Accessed 2021-04-27].

Dashboard, 2021. Available from: https://wandb.ai/jam244/cloud/sweeps [Accessed 2021-05-05].

Davies, E.R., 2012. *Computer and machine vision : Theory, algorithms, practicalities. 4th ed*. Web: Elsevier.

E, S., J, L. and Fully, D.T., 2017. Convolutional Networks for Semantic Segmentation. *Ieee trans pattern anal mach intell*, pp.640–651.

F1 score = dice coefficient, 2021. Available from: https://chenriang.me/2020/05/05/f1-equal-dice-coefficient [Accessed 2021-04-27].

Ghosh, S., 2019. Understanding Deep Learning Techniques for Image Segmentation. *Acm computing surveys 52.4*, pp.1–36.

Hyperparameter tuning for keras and pytorch models, 2021. Available from: https://wandb.ai/site/articles/hyperparameter-tuning-as-easy-as-1-2-3 [Accessed 2021-04-27].

Kaggle code, 2021. Available from: https://www.kaggle.com/c/understanding_cloud_organization/code [Accessed 2021-04-27].

Kaggle evaluation, 2021. Available from: https://www.kaggle.com/c/understanding_cloud_organization/overview/evaluation [Accessed 2021-04-27].

Kaggle leaderboard, 2021. Available from: https://www.kaggle.com/c/understanding_cloud_organization/leaderboard [Accessed 2021-04-27].

Kaggle submission, 2021. Available from: https://www.kaggle.com/dimitreoliveira/cloud-segmentation-with-utility-scripts-and-keras [Accessed 2021-04-27].

Kaiming, H., Georgia, G., Piotr, D. and Ross, G., 2020. Mask R-CNN. *Ieee transactions on pattern analysis and machine intelligence*, pp.386–97.

Krizhevsky, A., Sutskever, I. and Hinton, G.E., 2017. Imagenet classification with deep convolutional neural networks. *Commun. acm*, 60(6), p.84–90. Available from: https://doi.org/10.1145/3065386.

Olaf, R., Philipp, F. and Brox, T., 2015. U-Net: Convolutional Networks for Biomedical Image Segmentation. *Medical image computing and computer-assisted intervention – miccai 2015 9351*, pp.234–41.

Resnet34, 2021. Available from: https://models.roboflow.com/classification/resnet34 [Accessed 2021-05-07].

Scherr, T., Bartschat, A., Reischl, M., Stegmaier, J. and Mikut, R., 2018. Best practices in deep learning-based segmentation of microscopy images.

Segmentation models, 2021. Available from: https://github.com/qubvel/segmentation_models [Accessed 2021-04-27].

Understanding clouds from satellite images, 2021. Available from: https://www.kaggle.com/c/understanding_cloud_organization/ [Accessed 2021-04-27].

Understanding dice loss for crisp boundary detection, 2021. Available from: https://medium.com/ai-salon/understanding-dice-loss-for-crisp-boundary-detection-bb30c2e5f62b [Accessed 2021-04-27].

Website, 2021. Available from: https://wandb.ai/site [Accessed 2021-04-27].

9 Appendix

9.1 Appendix Subsection