Understanding Clouds from Satellite Images

Sprint 1 Goals

- Decide which image processing API we are going to use.
- Formulate a goal accuracy to aim for
- Create a working image analyzer for cloud types/density.
- Combine the API and our image analyzing program into one master program.

Steps Taken

We began to look at other Kaggle competitions to get a sense of where to begin. They were also instrumental in understanding how machine learning code works.

https://www.kaggle.com/c/titanic

The Kaggle Titanic competition was a great place to start to learn the basic concepts and strategies to solving machine learning problems.

Sprint 1 Outcomes

- Using Keras with Tensorflow backend
  - Why?
    - Higher level API: Very easy to use
    - Supports Multiple backends: Tensorflow, Theano, CNTK
    - Used heavily
- Score to aim for : 0.50 - 0.60
  - Why?
    - Highest Score in competition currently is : 0.677
    - Score is evaluated as: Dice Coefficient

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

X is the predicted set of pixels and Y is the ground truth

Sprint 2

- Results
- Demo
- System Architecture
- Major Components

Results

Results

- Dice coefficient score: 0.577 (*)
- Highest is 0.677

* We did not right all of the code. A **lot** of the code is common knowledge passed down between kaggle kernels.

Demo / Kaggle Notebook

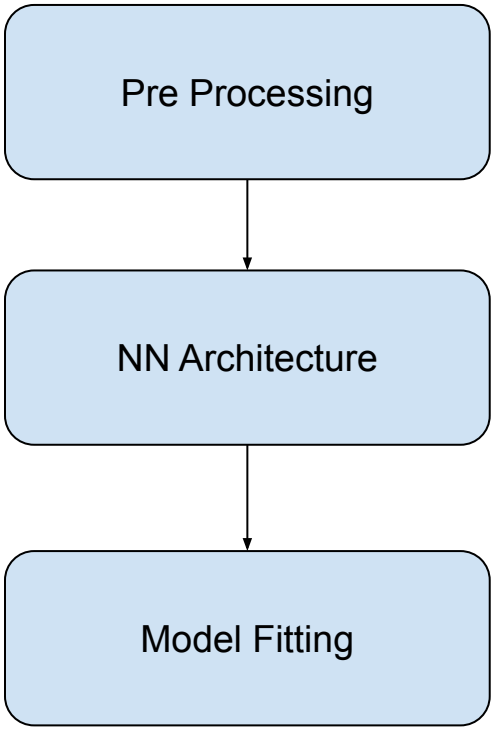
Demo / Kaggle Notebook

- Link to notebook:
  https://www.kaggle.com/arorashu/cloud-det-nb/data?scriptVersionId=22995066

High level System Architecture

---

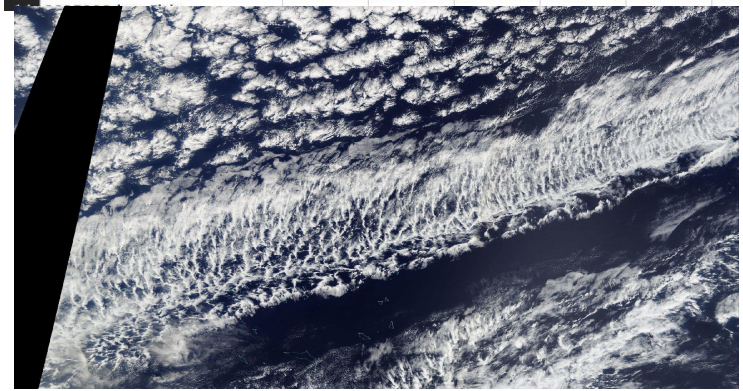
1. Preprocess Data
2. Define neural net architecture
3. Perform Training : Model Fitting

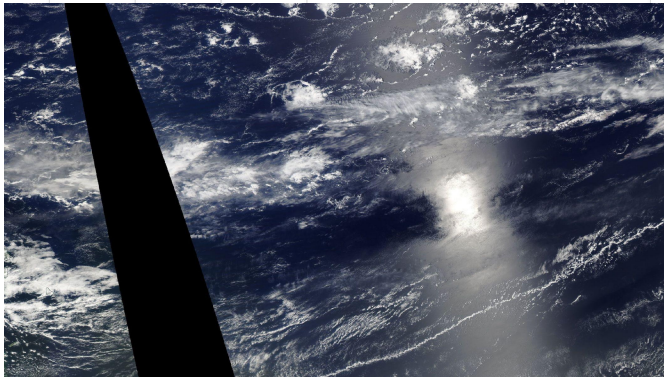


1. Preprocess Data

How the Input Data looks like

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	Image_Label	EncodedPixels															
2	0011165.jpg_Fish	264918 937 266318 937 267718 937 269118 937 270518 937 271918 937 273318 937 274718 937 276118 937 277518 937 278918 937 280318 937 281718 937 283118 937 28451															
3	0011165.jpg_Flower	1355565 1002 1356965 1002 1358365 1002 1359765 1002 1361165 1002 1362565 1002 1363965 1002 1365365 1002 1366765 1002 1368165 1002 1369565 1002 1370965 1002 1															
4	0011165.jpg_Gravel																
5	0011165.jpg_Sugar																
6	002be4f.jpg_Fish	233813 878 235213 878 236613 878 238010 881 239410 881 240810 881 242210 881 243610 881 245010 881 246410 881 247810 881 249210 881 250610 881 252010 881 25341															
7	002be4f.jpg_Flower	1339279 519 1340679 519 1342079 519 1343479 519 1344879 519 1346279 519 1347679 519 1349079 519 1350479 519 1351879 519 1353279 519 1354679 519 1356079 519 13															
8	002be4f.jpg_Gravel																
9	002be4f.jpg_Sugar	67495 350 68895 350 70295 350 71695 350 73095 350 74495 350 75895 350 77295 350 78695 350 80095 350 81495 350 82895 350 84295 350 85695 350 87095 350 88495 350 8															
10	0031ae9.jpg_Fish	3510 690 4910 690 6310 690 7710 690 9110 690 10510 690 11910 690 13310 690 14710 690 16110 690 17510 690 18910 690 20310 690 21710 690 23110 690 24510 690 25910 6															
11	0031ae9.jpg_Flower	2047 703 3447 703 4847 703 6247 703 7647 703 9047 703 10447 703 11847 703 13247 703 14647 703 16047 703 17447 703 18847 703 20247 703 21647 703 23047 703 24447 70															
12	0031ae9.jpg_Gravel																
13	0031ae9.jpg_Sugar	658170 388 659570 388 660970 388 662370 388 663770 388 665170 388 666570 388 667970 388 669370 388 670770 388 672170 388 673570 388 674970 388 676370 388 67777															





How the Input Data looks like

Training Images: 5546

3.46 GB

Test Images: 3698

2.3 GB

Labels: 5546 * 4 = 22184

200MB

Time for 10 epochs: around 2 hours - on a GPU!

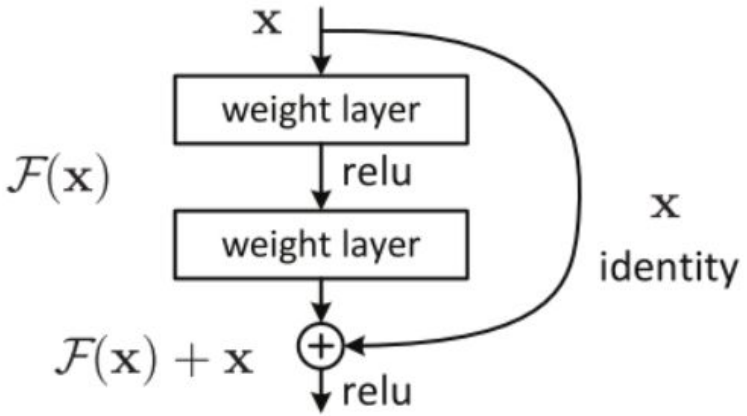
2. Define neural net architecture

ResNet

1. Utilizes skip connection

2. Most useful trick to get more performance out of deeper layers

3. We use the Standard ResNet-34



a residual block

Why ResNet?

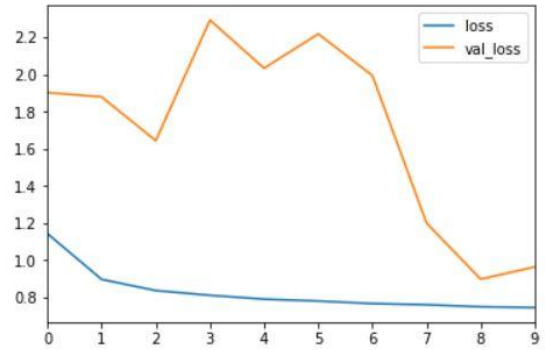
---

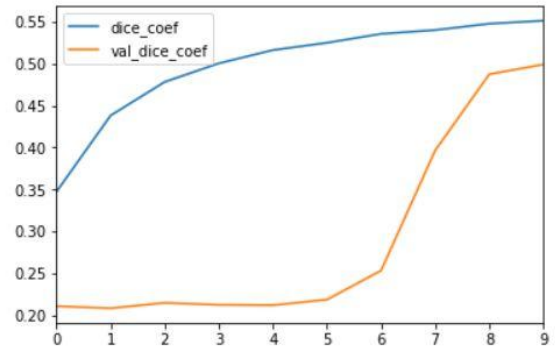
1. ResNet are the state of the art in Image Segmentation
2. Libraries like Tensorflow have very efficient implementations
3. ResNets, UNets and their variants are the most in this Kaggle competition

3. Perform Training : Model Fitting

Perform Training : Model Fitting

```
Epoch 1/10
138/138 [==============================] - 800s 6s/step - loss: 1.1424 - dice_coef: 0.3473 - val_loss: 1.9020 - val_dice_coef: 0.2107
Epoch 2/10
138/138 [==============================] - 814s 6s/step - loss: 0.8962 - dice_coef: 0.4381 - val_loss: 1.8795 - val_dice_coef: 0.2082
Epoch 3/10
138/138 [==============================] - 763s 6s/step - loss: 0.8361 - dice_coef: 0.4782 - val_loss: 1.6433 - val_dice_coef: 0.2147
Epoch 4/10
138/138 [==============================] - 826s 6s/step - loss: 0.8104 - dice_coef: 0.5003 - val_loss: 2.2932 - val_dice_coef: 0.2123
Epoch 5/10
138/138 [==============================] - 754s 5s/step - loss: 0.7897 - dice_coef: 0.5161 - val_loss: 2.0335 - val_dice_coef: 0.2119
Epoch 6/10
138/138 [==============================] - 819s 6s/step - loss: 0.7795 - dice_coef: 0.5249 - val_loss: 2.2183 - val_dice_coef: 0.2185
Epoch 7/10
138/138 [==============================] - 764s 6s/step - loss: 0.7658 - dice_coef: 0.5355 - val_loss: 1.9942 - val_dice_coef: 0.2532
Epoch 8/10
138/138 [==============================] - 755s 5s/step - loss: 0.7597 - dice_coef: 0.5400 - val_loss: 1.1998 - val_dice_coef: 0.3965
Epoch 9/10
138/138 [==============================] - 764s 6s/step - loss: 0.7486 - dice_coef: 0.5476 - val_loss: 0.8975 - val_dice_coef: 0.4873
Epoch 10/10
138/138 [==============================] - 739s 5s/step - loss: 0.7443 - dice_coef: 0.5512 - val_loss: 0.9633 - val_dice_coef: 0.4991
```





+ Code + Markdown