

Machine Learning Engineer Nanodegree

Capstone Project

Bhavani Chillapalli

September 30th, 2018

I. Definition

Project Overview

Several areas of Earth with large accumulations of oil and gas also have huge deposits of salt below the surface. But unfortunately, knowing where large salt deposits are precisely is very difficult. Professional seismic imaging still requires expert human interpretation of salt bodies. This leads to very subjective, highly variable renderings. More alarmingly, it leads to potentially dangerous situations for oil and gas company drillers.

Seismic data is collected using reflection seismology, or seismic reflection. The method requires a controlled seismic source of energy, such as compressed air or a seismic vibrator, and sensors record the reflection from rock interfaces within the subsurface. The recorded data is then processed to create a 3D view of earth's interior. Reflection seismology is similar to X-ray, sonar and echolocation.

A seismic image is produced from imaging the reflection coming from rock boundaries. The seismic image shows the boundaries between different rock types. In theory, the strength of reflection is directly proportional to the difference in the physical properties on either sides of the interface. While seismic images show rock boundaries, they don't say much about the rock themselves; some rocks are easy to identify while some are difficult.

TGS(the world's leading geoscience data company) teamed up with Kaggle's machine learning community to build an algorithm that automatically and accurately identifies if a subsurface target is salt or not.

The data is a set of images chosen at various locations chosen at random in the subsurface. The images are 101 x 101 pixels and each pixel is classified as either salt or sediment. In addition to the seismic images, the depth of the imaged location is provided for each image. The goal of the competition is to segment regions that contain salt.

Problem Statement

The main problem to be solved is to identify if a subsurface target is a salt. There are several areas of the world where there are vast quantities of salt in the subsurface. One of the challenges of seismic imaging is to identify the part of subsurface which is salt. Salt has characteristics that makes it both simple and hard to identify. Salt density is usually 2.14 g/cc which is lower than most surrounding rocks. The seismic velocity of salt is 4.5 km/sec, which is usually faster than its surrounding rocks.

This difference creates a sharp reflection at the salt-sediment interface. Usually salt is an amorphous rock without much internal structure. This means that there is typically not much reflectivity inside the salt, unless there are sediments trapped inside it. The unusually high seismic velocity of salt can create problems with seismic imaging.

Solution :

1. Data exploration
2. Building the Model
3. Hyperparameter Tuning

Metrics

This competition is evaluated on the mean average precision at different intersection over union (IoU) thresholds. The IoU of a proposed set of object pixels and a set of true object pixels is calculated as:

$$IoU(A, B) = \frac{A \cap B}{A \cup B}.$$

The metric sweeps over a range of IoU thresholds, at each point calculating an average precision value. The threshold values range from 0.5 to 0.95 with a step size of 0.05: (0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95). In other words, at a threshold of 0.5, a predicted object is considered a "hit" if its intersection over union with a ground truth object is greater than 0.5. At each threshold value t , a precision value is calculated based on the number of true positives (TP), false negatives (FN), and false positives (FP) resulting from comparing the predicted object to all ground truth objects:

$$\frac{TP(t)}{TP(t) + FP(t) + FN(t)}.$$

A true positive is counted when a single predicted object matches a ground truth object with an IoU above the threshold. A false positive indicates a predicted object had no associated ground truth object. A false negative indicates a ground truth object had no associated predicted object. The average precision of a single image is then calculated as the mean of the above precision values at each IoU threshold:

$$\frac{1}{|thresholds|} \sum_t \frac{TP(t)}{TP(t) + FP(t) + FN(t)}.$$

Lastly, the score returned by the competition metric is the mean taken over the individual average precisions of each image in the test dataset.

In my initial attempt i used a custom '[mean_iou](#)' metric. I wanted the mean average precision at different intersection over union(IoU) thresholds in Keras. Tensorflow has a mean IoU metric, but it doesn't support mean over multiple thresholds.

```
# Define IoU metric
def mean_iou(y_true, y_pred):
    prec = []
    for t in np.arange(0.5, 1.0, 0.05):
        y_pred_ = tf.to_int32(y_pred > t)
        score, up_opt = tf.metrics.mean_iou(y_true, y_pred_, 2)
        K.get_session().run(tf.local_variables_initializer())
        with tf.control_dependencies([up_opt]):
            score = tf.identity(score)
        prec.append(score)
    return K.mean(K.stack(prec), axis=0)
```

I later modified the code for my iou_metric as i went about refining my model :

```
def iou_metric(imgs_true, imgs_pred):
    num_images = len(imgs_true)
    scores = np.zeros(num_images)

    for i in range(num_images):
        if imgs_true[i].sum() == imgs_pred[i].sum() == 0:
            scores[i] = 1
        else:
            scores[i] = (iou_thresholds <= iou(imgs_true[i], imgs_pred[i])).mean()

    return scores.mean()
```

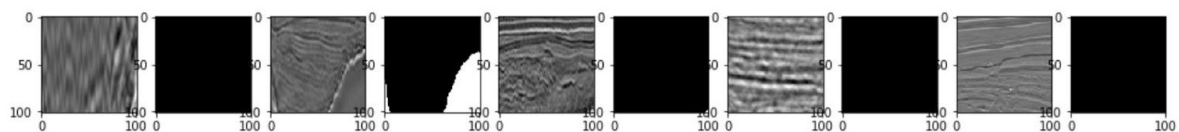
Intersection over union or IoU is a common metric for assessing performance in semantic segmentation tasks. In a sense, IoU is to segmentation what an F1 score is to classification.

Both are non-differentiable, and not normally optimized directly. Optimizing cross entropy loss is a common proxy for these scores, that usually leads to decent performance, provided everything else has been setup correctly, e.g regularization, stopping training at an appropriate time.

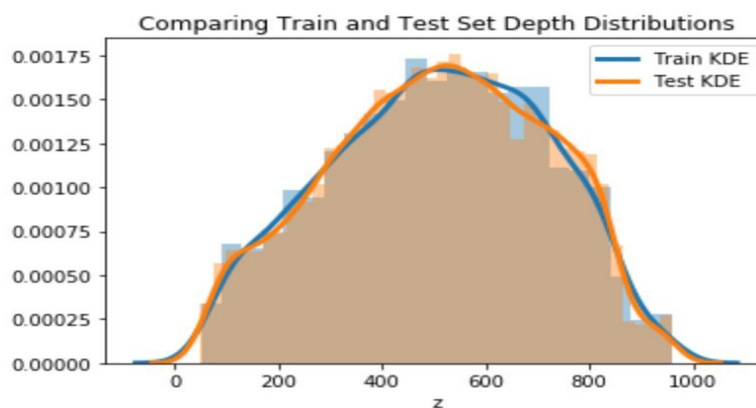
II. Analysis

Data Exploration

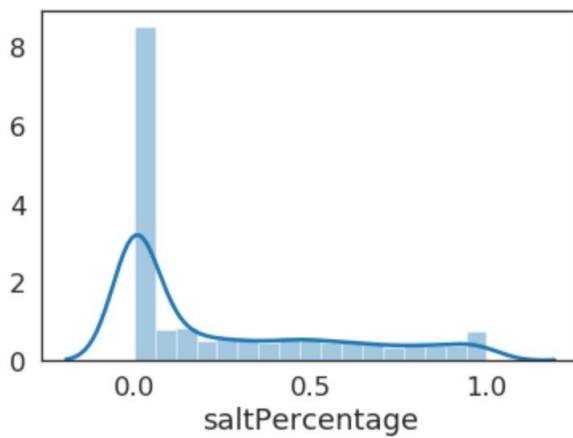
The data is a set of images with 101 x 101 pixels. Each pixel is either a sediment or a salt. We are also provided with the depths of each imaged location. There is also a helper file that shows the training set masks in run-length encoded format. After a basic exploration of the data , we notice that the images look like this :



There are some images with absolutely no salt. This is great as the algorithm we build will know that patches exist entirely without salt. I also ran a depth distribution of the images in both the training and testing sets.



Next, i tried to see if there is a correlation between depth and salt content.



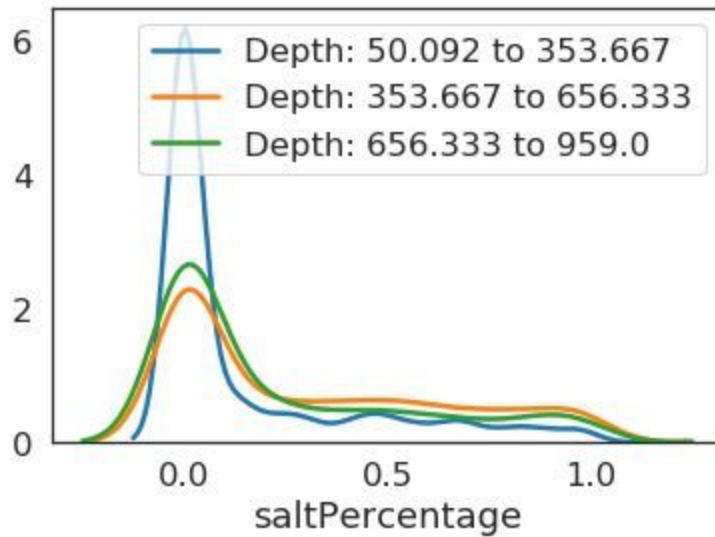
I found that there was a correlation of 0.1 between salt percentage and depth. So I decided to dig deeper.

```
bin
(50.092, 353.667]    0.154660
(353.667, 656.333]    0.304098
(656.333, 959.0]      0.237646
Name: saltPercentage, dtype: float64
```

```
Depth: 50.092 to 353.667
      saltPercentage      z
saltPercentage    1.000000  0.25739
z                0.25739  1.00000
Depth: 353.667 to 656.333
      saltPercentage      z
saltPercentage    1.000000 -0.014134
z                -0.014134  1.000000
Depth: 656.333 to 959.0
      saltPercentage      z
saltPercentage    1.000000 -0.039636
z                -0.039636  1.000000
```

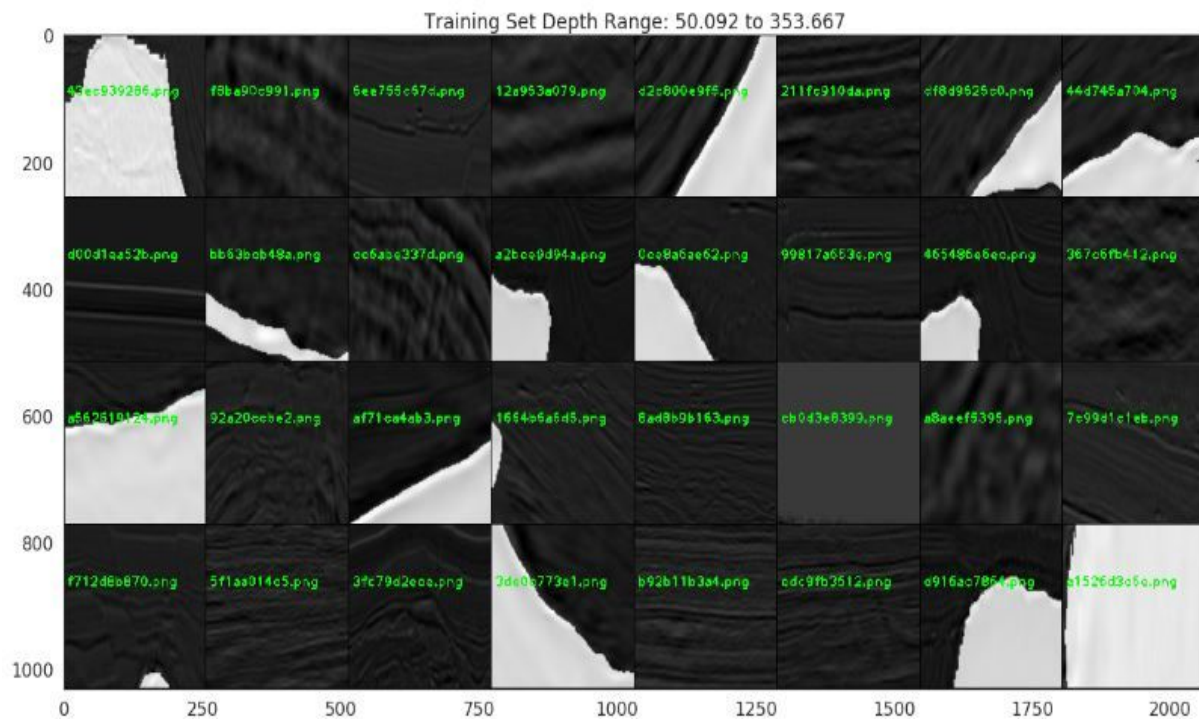
We can see that the mean Salt Percentage is highest for the group of images at depth between 353 and 656. We can further notice that correlation of Salt Percentage is high for depths

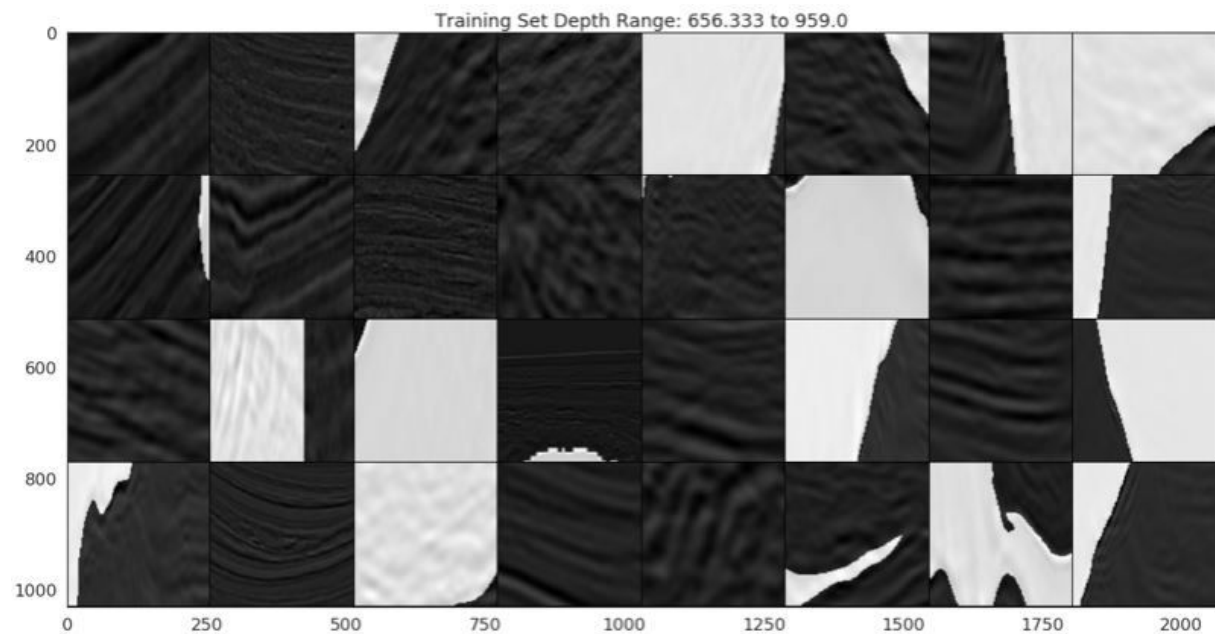
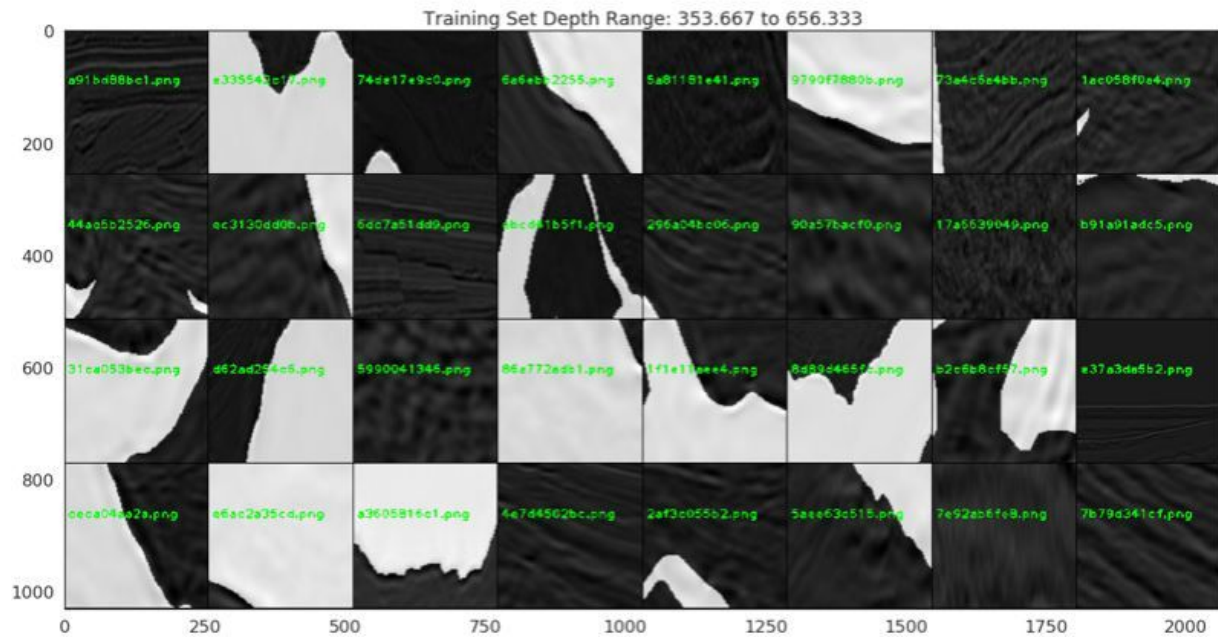
between 50 and 353 which may mean that as we go deeper from 50 to 353, the salt content seems to increase. At deeper depths, the correlation seems to be negligible.



We can see from above that there are lots of images with no salt content in the 50 to 353 depth bucket. It can also be observed that the bucket : 353 to 656 has the highest salt content for Salt Percentage greater than 0.25.

I plotted the images at different depths to see if there are any visible differences:

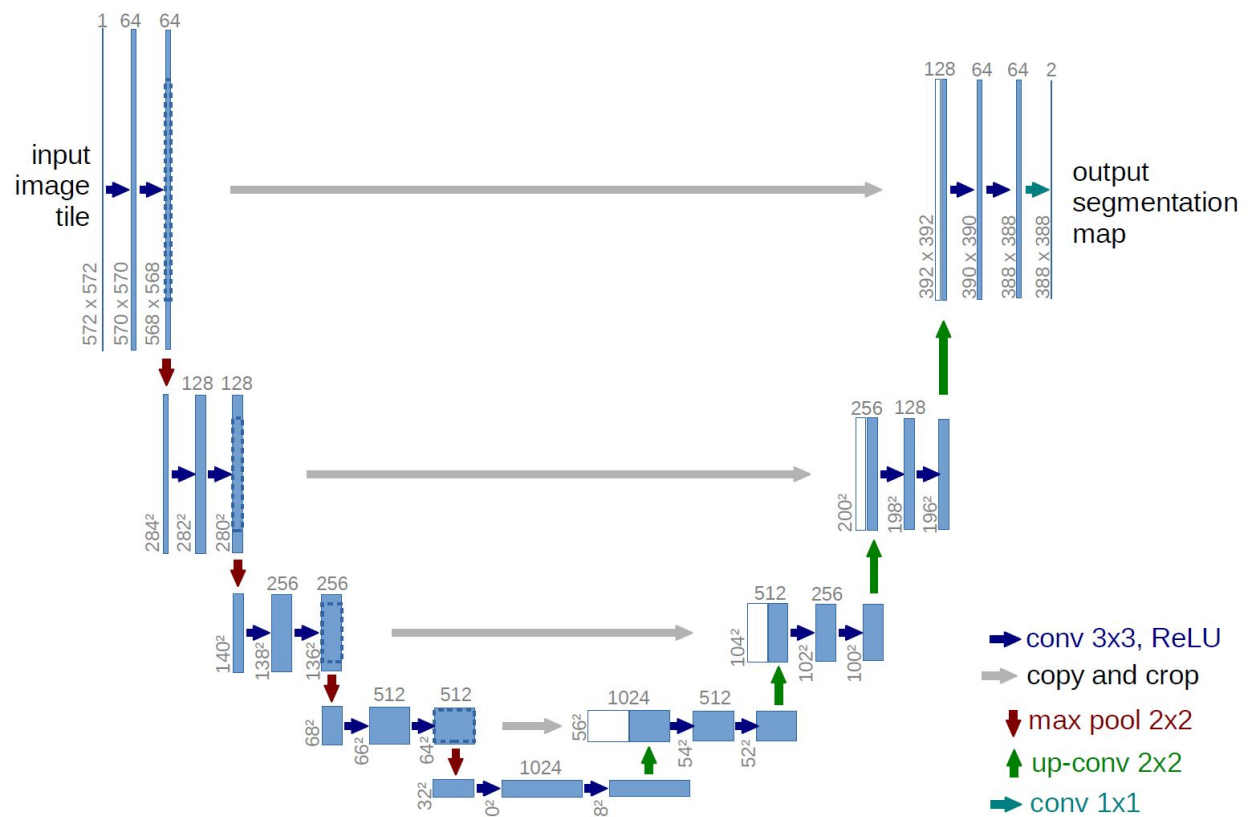




It seems that at depth 0 i.e. 50 to 353 the masks appear to be in the lower part of the image. Similarly at depth 2 i.e. 656 to 959 the masks appear to be in the lower part of the image. I plotted the mean depth of the masks and found that salt is not equally distributed between all depths.

Algorithms and Techniques

For this particular type of image segmentation problem , i thought it was appropriate to use a U-Net Model.It consists of a contracting path (left side) and an expansive path (right side).



- The contracting path follows the typical architecture of a convolutional network. It consists of the repeated application of two 3×3 convolutions, each followed by a batch normalization layer and a rectified linear unit (ReLU) activation and dropout and a 2×2 max pooling operation with stride 2 for downsampling. At each downsampling step we double the number of feature channels. The purpose of this contracting path is to capture the context of the input image in order to be able to do segmentation.
- Every step in the expansive path consists of an upsampling of the feature map followed by a 2×2 convolution (“up-convolution”) that halves the number of feature channels, a concatenation with the correspondingly feature map from the contracting path, and two 3×3 convolutions, each followed by batchnorm, dropout and a ReLU. The purpose of this

expanding path is to enable precise localization combined with contextual information from the contracting path.

- At the final layer a 1×1 convolution is used to map each 16- component feature vector to the desired number of classes.

We can experiment with this model in conjunction with different kinds of Convolutional Layers/drop out layers , optimizers and losses to arrive at the best model for our problem.

Layers :

- **Convolution** : Convolutional layers convolve around the image to detect edges, lines, blobs of colors and other visual elements. Convolutional layers hyperparameters are the number of filters, filter size, stride, padding and activation functions for introducing non-linearity.
- **MaxPooling** : Pooling layers reduces the dimensionality of the images by removing some of the pixels from the image. Maxpooling replaces a $n \times n$ area of an image with the maximum pixel value from that area to downsample the image.
- **Dropout** : Dropout is a simple and effective technique to prevent the neural network from overfitting during the training. Dropout is implemented by only keeping a neuron active with some probability p and setting it to 0 otherwise. This forces the network to not learn redundant information.

Activation functions :

Activation layers apply a non-linear operation to the output of the other layers such as convolutional layers or dense layers.

- **ReLU Activation** : ReLu or Rectified Linear Unit computes the function $f(x)=\max(0,x)$ to threshold the activation at 0.
- **Softmax Activation** : [Softmax function](#) is applied to the output layer to convert the scores into probabilities that sum to 1.

Optimizers :

- **Adam** : [Adam](#) (Adaptive moment estimation) is an update to RMSProp optimizer in which the running average of both the gradients and their magnitude is used. In practice Adam is currently recommended as the default algorithm to use, and often works slightly better than RMSProp. In my experiments Adam also shows general high accuracy while

adadelata learns too fast. I've used Adam in all the experiments because I felt having similar optimizer would be a better baseline for comparing the experiments.

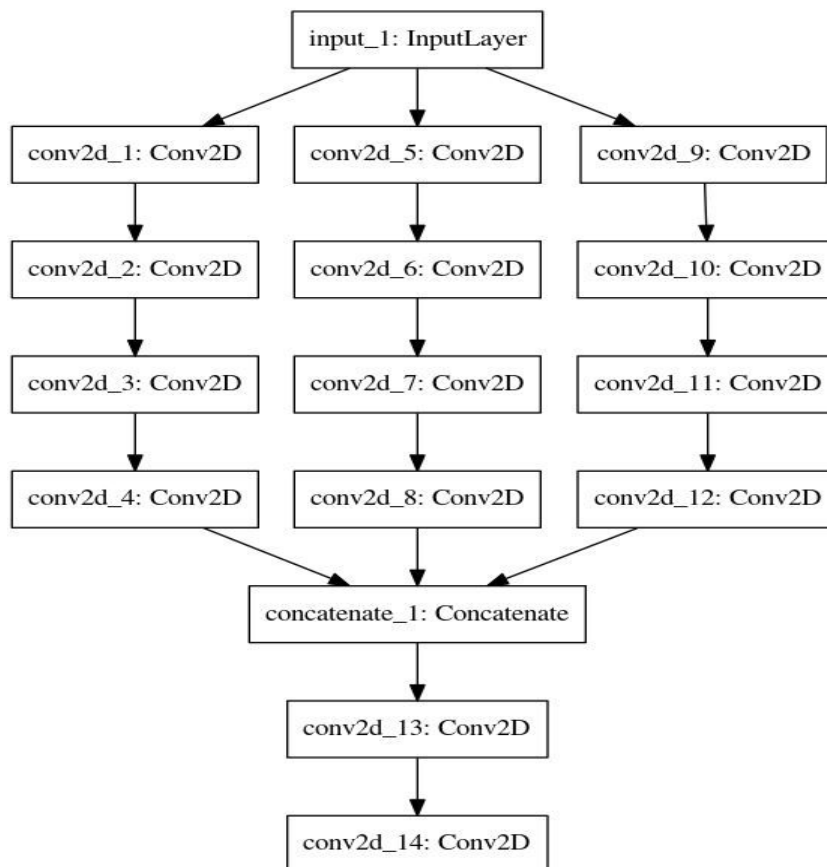
Data Augmentation : Data augmentation is a regularization technique where we produce more images from the training data provided with random jitter, crop, rotate, reflect, scaling etc to change the pixels while keeping the labels intact. CNNs generally perform better with more data as it prevents overfitting.

Batch Normalization : Batch Normalization is a A recently developed technique by Ioffe and Szegedy which tries to properly initialize neural networks by explicitly forcing the activations throughout a network to take on a unit gaussian distribution at the beginning of the training. In practice we put the Batchnorm layers right after Dense or convolutional layers. Networks that use Batch Normalization are significantly more robust to bad initialization. Because normalization greatly reduces the ability of a small number of outlying inputs to over-influence the training, it also tends to reduce overfitting. Additionally, batch normalization can be interpreted as doing preprocessing at every layer of the network, but integrated into the network itself.

Benchmark

A basic benchmark model is implemented using convolutional blocks. The model was compiled using the Adam's optimizer and binary Cross Entropy Loss.

This is the baseline model's architecture:



It achieved a Score of 0.23. My goal is to fine tune this base model to arrive at a higher efficiency for salt detection in seismic images. A well-designed convolutional neural network should be able to beat the random choice baseline model easily.

III. Methodology

Data Preprocessing

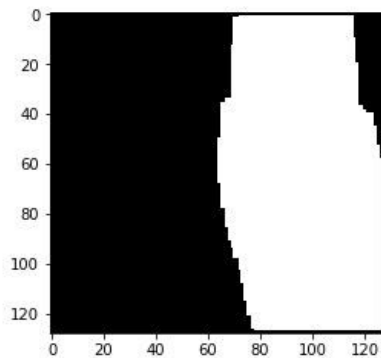
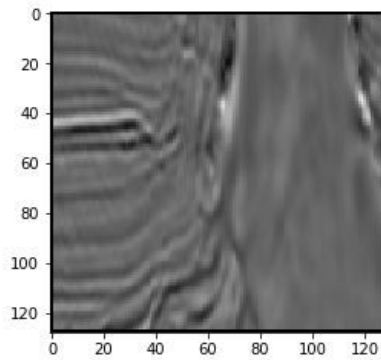
In my first attempt , i re-sized the images before they were fed to the U-Net. I wanted to experiment with this idea to see if it plays any role in efficiency. I implemented several versions with and without resizing and eventually found that i was losing too much information while resizing. So the most optimal result was obtaining using the actual size of images.

Other than this experiment i did not have to do any other pre processing. The images were clean and were ready to be fed to the U-Net.

Implementation

My first step was to inspect the data. Upon some basic data exploration i found that the data is varied. This was really important if i wanted the algorithm to generalize well. I found that there were lots of samples without salt. So the data was not biased at all. This made my task much easier.

My next step was resizing the images. This was only data pre processing that was required. It is always considered best to resize the images to a multiple of 2. After resizing i checked to see if the training data looked alright.



My next task was to train and build the model. After i created a function for the mean_iou metric used to validate the model, i focussed on building the sequential model. I added early stopping to the model to prevent overfitting.

The implementation of the model follows the U-Net architecture. My model consisted of Conv2D layers and MaxPooling2D layers. I used the 'relu' activation function on the input layers.. I used a sigmoid activation function on my final output layer. Adam's Optimizer was used with binary Cross Entropy loss to compile this model.

The parameters passed to the early stopper were patience = 5 , verbose = 1. I also created a check pointer to save the best model in each epoch. The model is run on the testing set with validation_split = 0.1 , batch_size = 8 , epochs = 30. The metric used to compile this model is mean_iou.

After the model is compiled, the next step is to resize the testing data just like we did the training data. This takes some time as there are over 18000 samples. Now we perform image

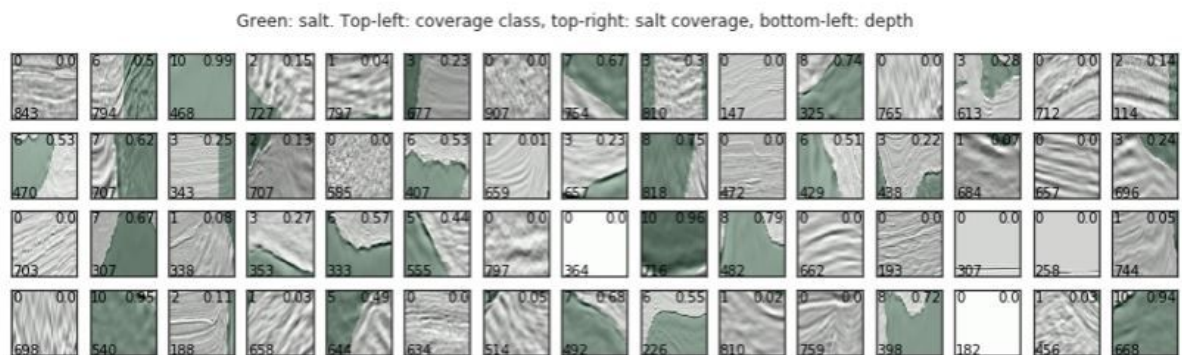
segmentation using the training set, validation set on the testing set. The final output file has an image id and rle_mask. The rle_mask consists of pixels that contain salt.

The most challenging aspect of writing this implementation was the iou metric. Building the sequential model is quite simple because of the vast array of information available on U-Nets and their architecture. I experimented with the architecture and was able to arrive at an optimal solution. Another concept that is important is the upsampling of images. Understanding the relevance of this component is extremely crucial to solving image segmentation problems effectively. Upsampling refers to increasing the resolution of images to get rid of gaps of any kind. This kind of scaling may help improve the performance of the model.

Refinement

I further refined this model by performing stratification and augmentation. I performed stratification by determining salt coverage classes. I calculated salt coverage by Counting the number of salt pixels in the masks and dividing them by the image size. I also created 11 coverage classes, -0.1 having no salt at all to 1.0 being salt only.

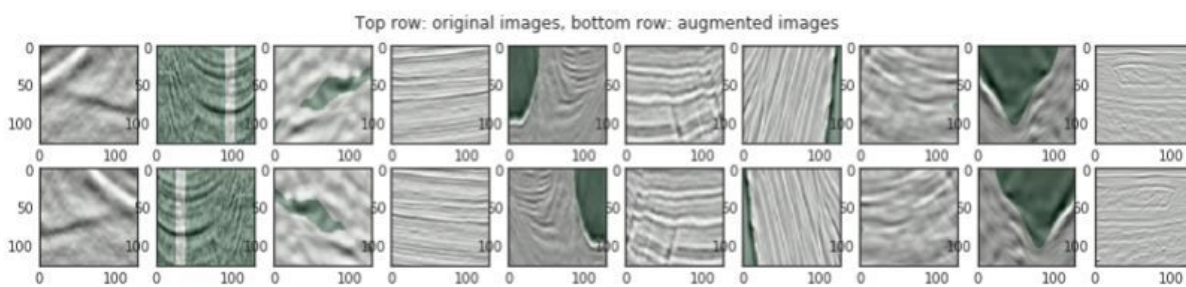
Used this stratification criterion to create some sample images like this :



I created a validation split of the data based on this stratification methodology. I made simple changes to the model. Apart from a Max Pooling Layer, I also added a Dropout Layer to the model. I used a sigmoid activation function in the output layer of the U-Net instead of a relu

activation function like my first model. I compiled the model with an Adam's optimizer and binary cross entropy loss. The metric used to validate the model was accuracy.

I also used the augmentation technique to train the images so that the model's performance is better. In order to combat the high expense of collecting thousands of training images, image augmentation has been developed in order to generate training data from an existing dataset. Image Augmentation is the process of taking images that are already in a training dataset and manipulating them to create many altered versions of the same image.



I ran this model for 200 epochs with a batch size of 32. It yielded a IoU score of 0.75 . This is a vast improvement from the my initial model.

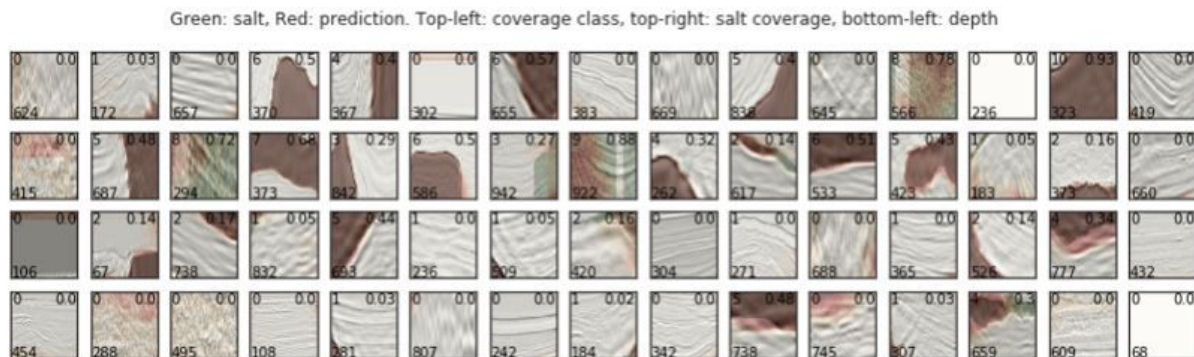
In my next attempt at refining the model i used resnet blocks in my U-Net. I also did not resize the images before feeding them to the U-Net. I replaced the Conv2D layers with resnet blocks. Resnets are known to be highly efficient at training CNN's with many layers. I experimented with a resnet in my model and obtained an IoU score of 0.79.

IV. Results

Model Evaluation and Validation

My final model proved to be quite robust when subjected to data augmentation. This model was derived from a basic U-Net model. I added and dropped layers to the model to arrive at this final improved version. The stratification and augmentation techniques made a huge difference to the model's performance.

I also performed a sanity check to validate the model's robustness.



This model is robust enough for the given problem. However, there is room for improvement. I tested this model with various testing sets and it yielded satisfying results.

Model	IoU Score
Random Choice (benchmark)	0.23
Basic U-Net Model with Image resizing	0.63
U-Net Model with stratification + Augmentation	0.75
U-Net Model with Resnet Blocks	0.79


Justification

Each model that i implemented was an improvement from the previous one. There was a vast difference in performance between the baseline model and the U-Net Model.I eventually implemented a U-Net with ResNet blocks.

868

▲141

BhavaniChillapalli




0.789

4

2d

Your Best Entry ▲

Your submission scored 0.789, which is an improvement of your previous score of 0.756. Great job!

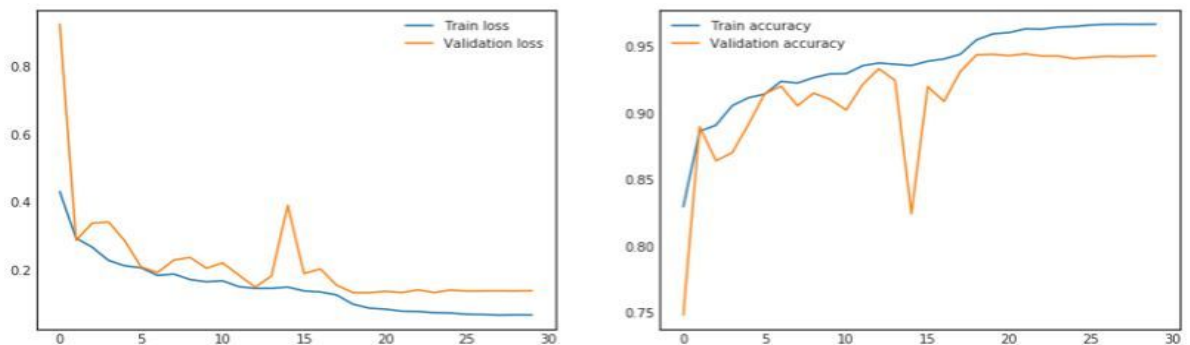
 Tweet this!

This was my best entry in the Kaggle leaderboard. I was also in the Top 32% at the time. However due to time constraints and expensive ,time consuming computations i wasn't able to experiment beyond this. I have some ideas to further improve the efficiency of the algorithm, which i will discuss below.

As data augmentation was used to train this model, it can also handle slight variations in the images such as horizontal flip, different illuminations, rotations and shifting up and down which are likely scenarios in real life. In the plot of the accuracy and loss for this model per epoch, it's also seen that the training accuracy/loss is converging with the validation one per epoch(reproduction and further comparison on that in the free-form visualization section). I've ran the model for around 5/6 hours for training where each epoch was taking me around 1 hour. With a good GPU I'd probably be able improve the accuracy by simply running the model for a few more epochs.

V. Conclusion

Free-Form Visualization



As we can see the training accuracy is near 100% in the diagram and the loss is near 0. Similarly the validation accuracy is also near 95% while the validation loss is around 0.2% near

the end of the 30 epochs. We also see the trend where the validation loss keeps decreasing initially but after around 2 epochs training loss keeps decreasing/accuracy keeps increasing, while the validation loss keeps increasing instead of decreasing.

This model's performance on the test set in the leaderboard is 0.79, which is better than the benchmark model's performance.

Reflection

To arrive at this end-to-end solution, I've tried to progressively use more complex models to classify the images. The baseline convolutional model heavily underperformed as it was oversimplistic and used very few layers. However, it was still a good starting point. Eventually the techniques that made the most difference to my model were image stratification and image augmentation. A U-Net model with Max Pooling Layers and Drop Out layers worked better in conjunction with those techniques. Since the images were trained with augmentation there was very less possibility of overfitting. This is why I believe that this is a pretty robust model. The best model was achieved with resnet blocks in the U-Net Model. After reading through discussion forums on Kaggle and doing some more research, I arrived at experimenting with resnet blocks.

The most difficult part for me was to get the experiments running on my local machine. Higher computational time results in lower number of experiments when it comes to neural networks, especially when I'm just figuring out what to do as it's my first experience with deep learning. However as I feel more confident in my skills in using deep learning now, I'll definitely try to seek more computational resources from now on.

Improvement

Due to time constraints and low computational power, it wasn't possible for me to experiment with other techniques. One idea I would like to experiment with in the future is using the Binary Lovasz loss function. I would also like to augment the data by performing random flips and 90 degree rotations. I hypothesize that this could improve the model's performance. I also want to try using a pretrained model and experiment with that.

References :

1. <https://keras.io/losses/>
2. <https://towardsdatascience.com/an-overview-of-resnet-and-its-variants-5281e2f56035>
3. <https://www.kaggle.com/vijaybj/basic-u-net-using-tensorflow>
4. <https://www.kaggle.com/keegil/keras-u-net-starter-lb-0-277>
5. <https://angusg.com/writing/2016/12/28/optimizing-iou-semantic-segmentation.html>