Machine Learning Engineer Nanodegree

TGS Salt Identification Challenge(Kaggle)

Amartya Kalapahar
December 24th, 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.
To create the most accurate seismic images and 3D renderings, TGS (the world's leading geoscience data company) is hoping Kaggle's machine learning community will be able to build an algorithm that automatically and accurately identifies if a subsurface target is salt or not.

Seismic data is a neat thing. We can imagine it like an ultra-sound of the subsurface. However, in an ultra-sound, we use much smaller wavelengths to image our body. Seismic data usually has wavelengths around 1m to 100m. That has some physical implications, but for now, we don't have to deal with that. It's just something to keep in mind while thinking about resolution.
Imaging salt has been a huge topic in the seismic industry, basically since they imaged salt the first time. The Society of Exploration geophysicist alone has over 10,000 publications with the keyword salt. Salt bodies are important for the hydrocarbon industry, as they usually form nice oil traps. So there's a clear motivation to delineate salt bodies in the subsurface. Seismic data interpreters are used to interpreting on 2D or 3D images that have been heavily processed. The standard work of seismic data analysis is open access. You'll find sections on Salt in there as well (https://wiki.seg.org/wiki/Salt-flank_reflections and https://wiki.seg.org/wiki/Salt_flanks). The seismic itself is pretty "old" in the publication, and you're dealing with data that is less noisy here, which is nice.

Problem Statement:

To create the most accurate seismic images and 3D renderings, TGS (the world's leading geoscience data company) is hoping Kaggle's machine learning community will be able to build an algorithm that automatically and accurately identifies if a subsurface target is salt or not.

The goal is to segment regions that contain salt.

A deep learning algorithm will be developed using Tensorflow/Keras and will be trained with training data. Specifically a CNN will be implemented in Tensorflow/Keras. Predictions will be made on the test data set and will be evaluated.

More specifically we will be using a unit model since it's good for segmentation type of problems.

Metrics:

Scoring 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.

Explanation of scoring metrics:

The metric used for this competition is defined as **the mean average precision at different intersection over union (IoU) thresholds**.

These are the steps to calculate to correct score:

1. For all of the images/predictions
    - Calculate the Intersection of Union metric ("compare" the original mask with your predicted one)
    - Calculate whether your predicted mask fits at a range of IoU thresholds.
    - At each threshold, "calculate" the precision of your predicted masks.
    - Average the precision across thresholds.
2. Across the dataset
    - Calculate the mean of the average precision for each image.

Picking test image

I'm going to pick a sample image from the training dataset, load the masks, then create a "mock predict"

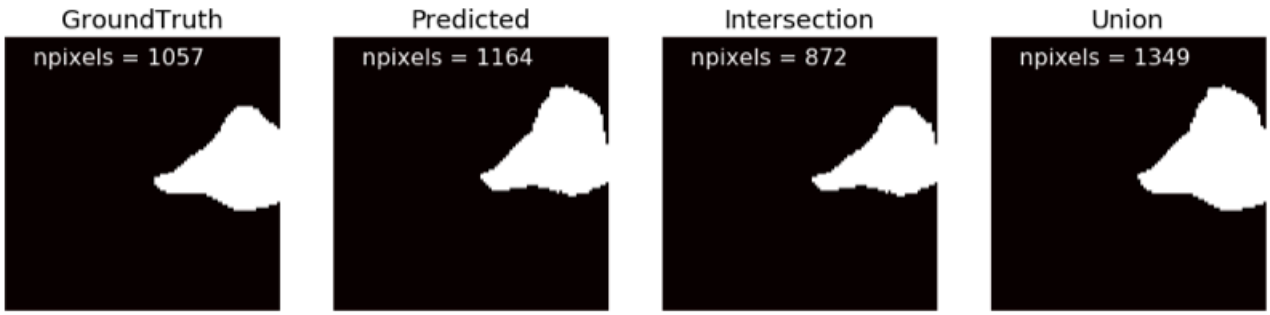
Code

Intersection Over Union (for a single Prediction-GroundTruth comparison)

> 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 intersection and the union of our GT and Pred masks are look like this:

Code



So, for this mask the **IoU** metric is calculated as:

$$IoU(A,B) = \frac{A \cap B}{A \cup B} = \frac{872}{1349} = 0.6464$$

Thresholding the IoU value (for a single GroundTruth-Prediction comparison)

> Next, we sweep over a range of IoU thresholds to get a vector for each mask comparison. 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.

Code

```
Does this IoU hit at each threshold?
0.50     True
0.55     True
0.60     True
0.65    False
0.70    False
0.75    False
0.80    False
0.85    False
0.90    False
0.95    False
Name: GT-P, dtype: bool
```

Single-threshold precision for a single image

> 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

I think this step causes the confusion. I think the evaluation description of this competition was simply copied from the desc of the 2018 Data Science Bowl (or something similar) competition, where you can predict more than one mask/object per image.

According to @William Cukierski's Discussion thread:

> Although this competition metric allows prediction of multiple segmented objects per image, note that we have encoded masks as one object here (see train.csv as an example). You will be best off to predict one mask per image.

Because of we have to predict only one mask per image, the computation below (from the evaluation desc) is a bit confusing:

> $$Precision(t) = \frac{TP(t)}{TP(t) + FP(t) + FN(t)}$$

This only makes sense if we would have more than one predicted mask/segment per image. In our case I think this would be a better description:

- GT mask is empty, your prediction non-empty: **(FP) Precision(threshold, image) = 0**
- GT mask non empty, your prediction empty: **(FN) Precision(threshold, image) = 0**
- GT mask empty, your prediction empty: **(TN) Precision(threshold, image) = 1**
- GT mask non-empty, your prediction non-empty: **(TP) Precision(threshold, image) = IoU(GT, pred) > threshold**

See @William Cukierski's comment here

> If a ground truth is empty and you predict nothing, you get a perfect score for that image. If the ground truth is empty and you predict anything, you get a 0 for that image.

Multi-threshold precision for a single image

> The average precision of a single image is then calculated as the mean of the above precision values at each IoU threshold:
>
> $$Avg.\,Precision = \frac{1}{n_{thresh}} \sum_{t=1}^{n} precision(t)$$

Here, we simply take the average of the precision values at each threshold to get our mean precision for the image.

Code

```
Average precision value for image `2bf5343f03` is 0.3
```

Let's see what happens if our prediction is 100% accurate.

```
Does this IoU hit at each threshold?
0.50    True
0.55    True
0.60    True
0.65    True
0.70    True
0.75    True
0.80    True
0.85    True
0.90    True
0.95    True
Name: GT-P, dtype: bool
Average precision value for image `2bf5343f03` is 1.0
```

Mean average precision for the dataset

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

Therefore, the leaderboard metric will simply be the mean of the precisions across all the images.

II. Analysis:

Data Exploration:

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.

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.

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 dataset can be downloaded from here : https://www.kaggle.com/c/10151/

download-all

The dataset contains 3 csv files and 2 zip files namely:

1. depths.csv : This file contains the depth underground (in feet) of each image. It has 2 columns namely id(A unique image identifier) and z(depth in feet). There are in total 22001 unique values in this file.

2. train.zip : This file contains the set of images required for training purpose. It includes both image and salt segmentation mask.

3. test.zip : This file contains set of images required for prediction.

4. train.csv : A helper file that shows the training set masks in run-length encoded format. It has two columns namely id(a unique image identifier) and rle_mask. There are in total 4001 unique values in the file.

5. sample_submission.csv : A sample submission showing the correct submission format. This file also contains 2 columns namely id(a unique image identifier) and rle_mask.

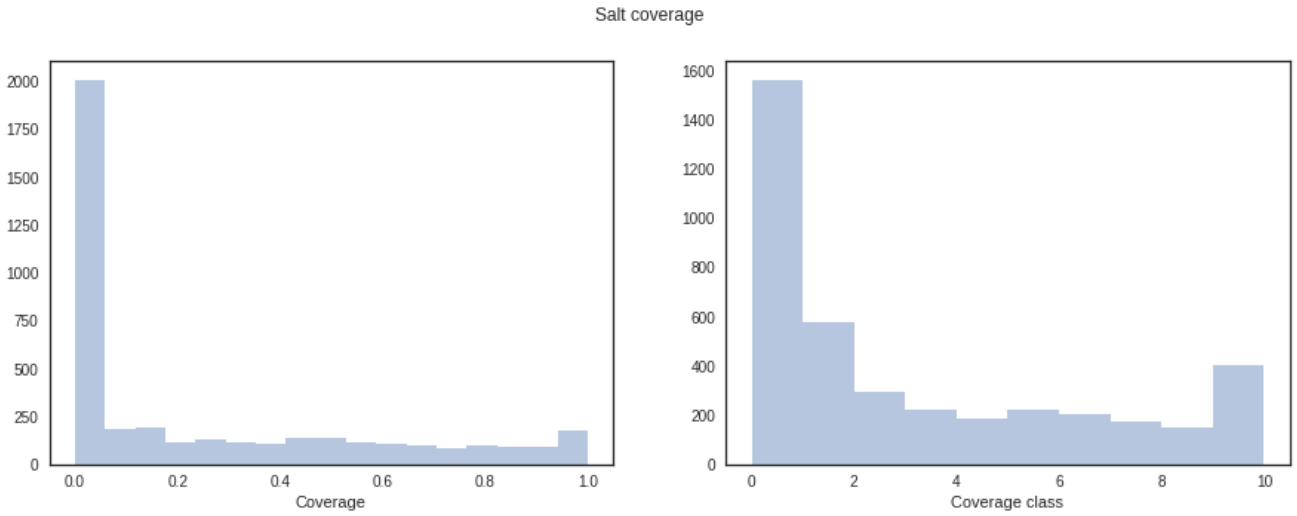
Printing 5 rows from the data

```
[ ]  print(train_df.head(5))
     print(test_df.head(5))
     len(train_df)
```

```
                 z
id
575d24d81d  843
a266a2a9df  794
75efad62c1  468
34e51dba6a  727
4875705fb0  797
                 z
id
353e010b7b  264
5439dbbddf  557
71bab9f311  846
52551f7a80  610
512d8d9997  577
4000
```

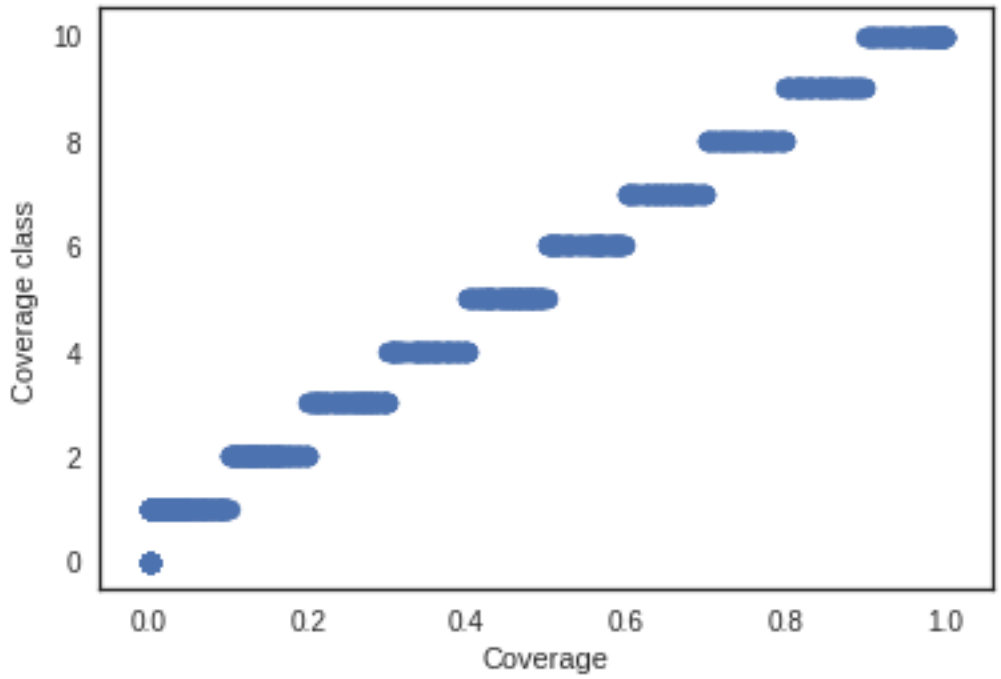
Exploratory Visualization:

Salt coverage and salt coverage classes:

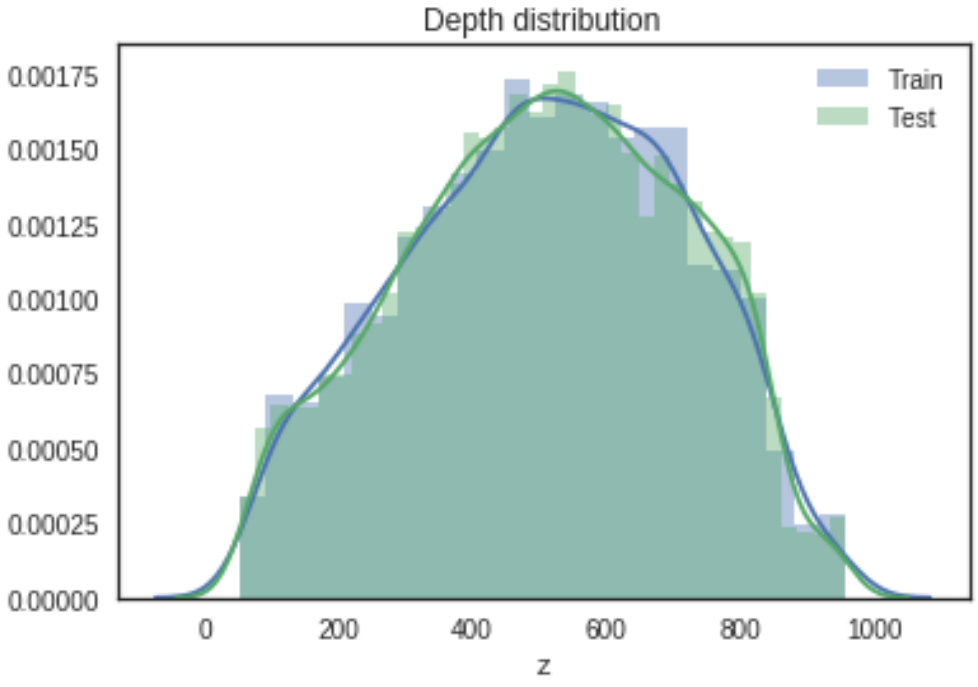


Counting the number of salt pixels in the masks and dividing them by the image size. Also create 11 coverage classes, -0.1 having no salt at all to 1.0 being salt only. Plotting the distribution of coverages and coverage classes, and the class against the raw coverage.

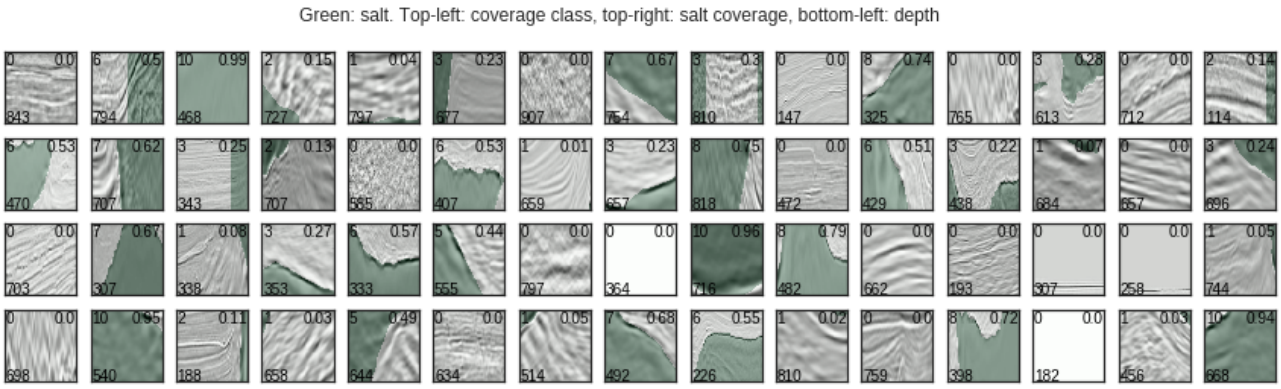
Salt Coverage Classes:



Plotting the depth distributions for the training and the testing data



Visualization of some images from the dataset with coverage class.

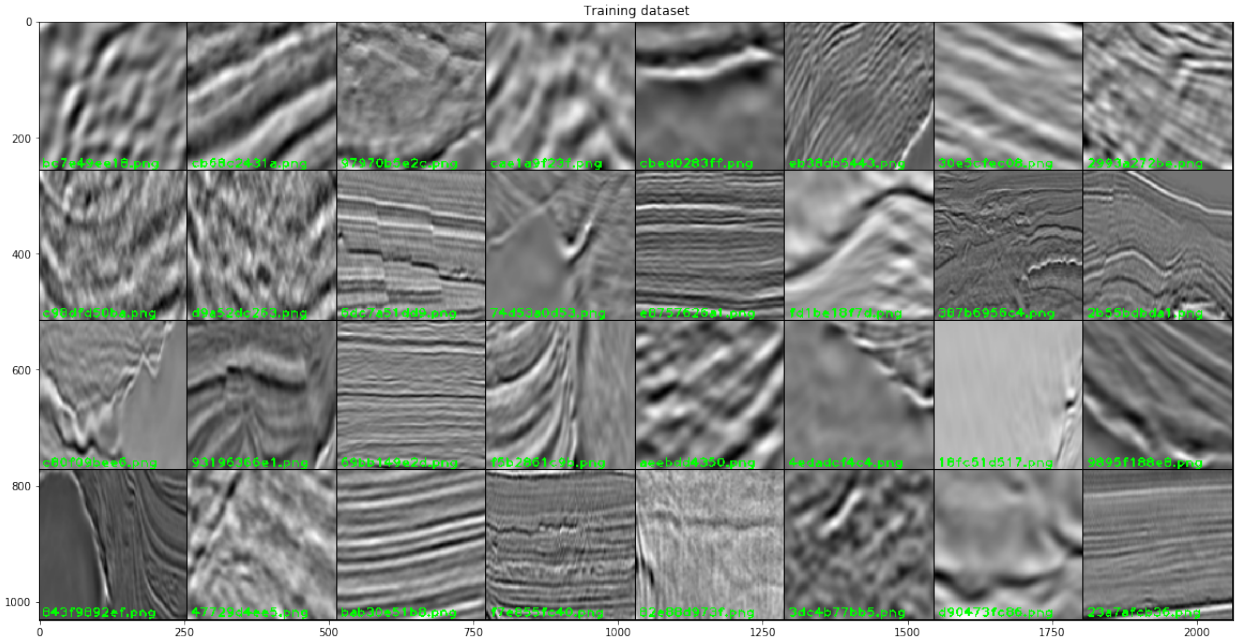


In the above image:

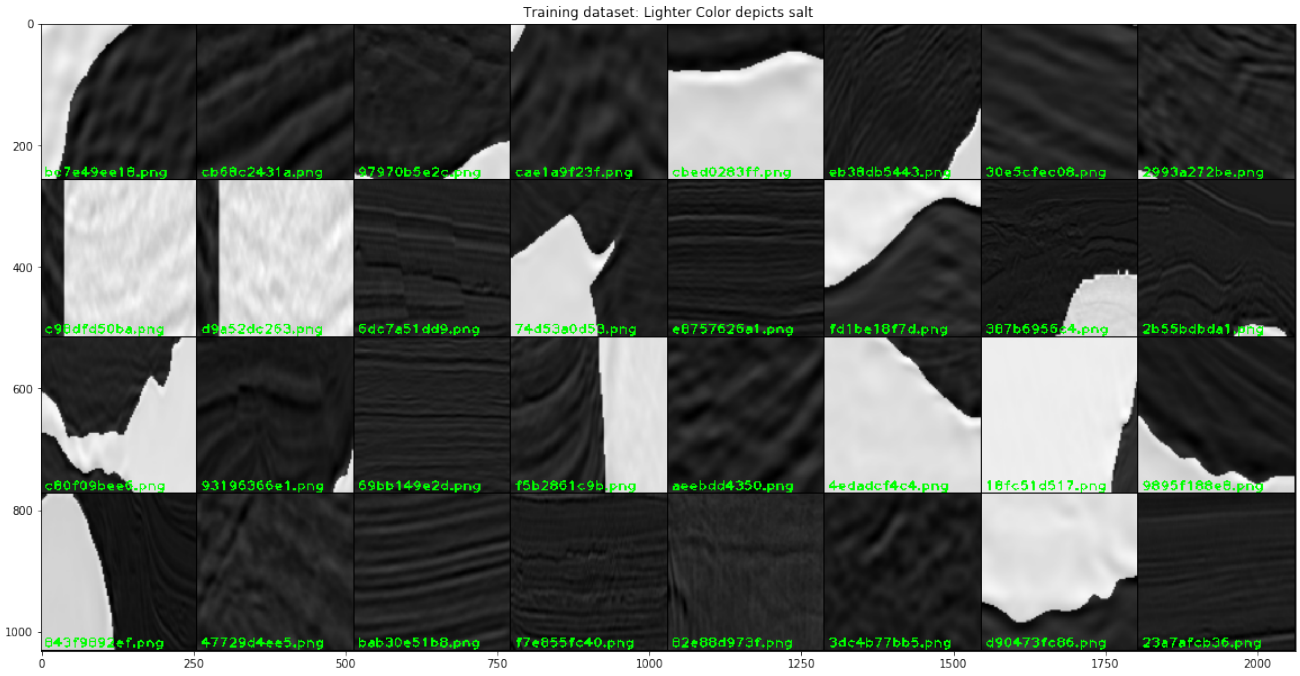
- Green represents salt
- Top-left represents coverage class
- Top right represents salt coverage
- Bottom left represents depth

Salt coverage is very important here since it will be used as a stratification criterion.

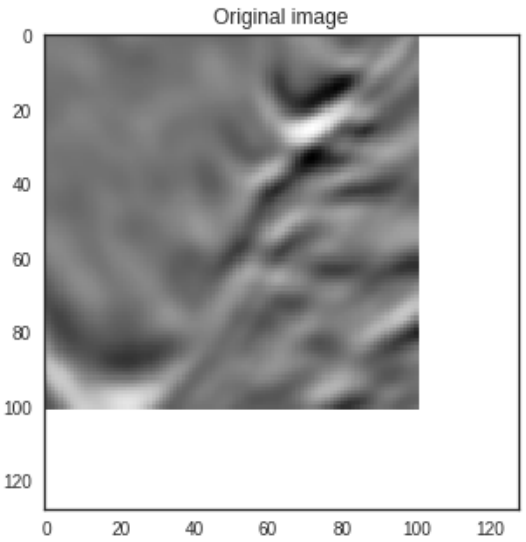
Some random images from the training dataset.

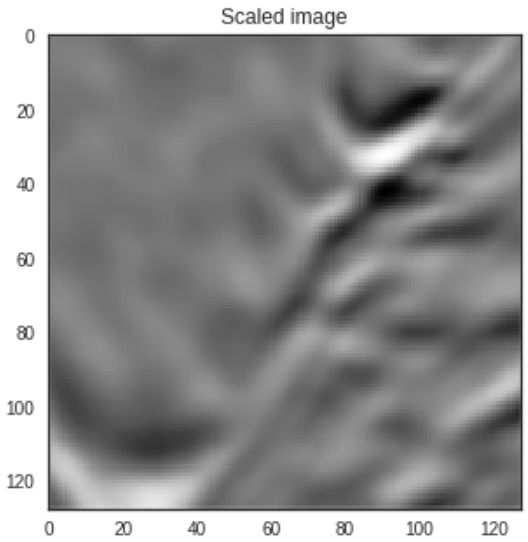


Training dataset(Lighter colour represents salt)



Scaling the images to fit the proper dimensions:





Algorithms and Techniques:

We will use a **Unet model architecture** here since it's good for segmentation type of problems. A diagram of the model is given below.

```
# --- make layer ---
# left
l1_1 = conlayer_left(3,1,3)
l1_2 = conlayer_left(3,3,3)
l1_3 = conlayer_left(3,3,3)

l2_1 = conlayer_left(3,3,6)
l2_2 = conlayer_left(3,6,6)
l2_3 = conlayer_left(3,6,6)

l3_1 = conlayer_left(3,6,12)
l3_2 = conlayer_left(3,12,12)
l3_3 = conlayer_left(3,12,12)

l4_1 = conlayer_left(3,12,24)
l4_2 = conlayer_left(3,24,24)
l4_3 = conlayer_left(3,24,24)

l5_1 = conlayer_left(3,24,48)
l5_2 = conlayer_left(3,48,48)
l5_3 = conlayer_left(3,48,24)

# right
l6_1 = conlayer_right(3,24,48)
l6_2 = conlayer_left(3,24,24)
l6_3 = conlayer_left(3,24,12)

l7_1 = conlayer_right(3,12,24)
l7_2 = conlayer_left(3,12,12)
l7_3 = conlayer_left(3,12,6)

l8_1 = conlayer_right(3,6,12)
l8_2 = conlayer_left(3,6,6)
l8_3 = conlayer_left(3,6,3)

l9_1 = conlayer_right(3,3,6)
l9_2 = conlayer_left(3,3,3)
l9_3 = conlayer_left(3,3,3)

l10_final = conlayer_left(3,3,1)
```

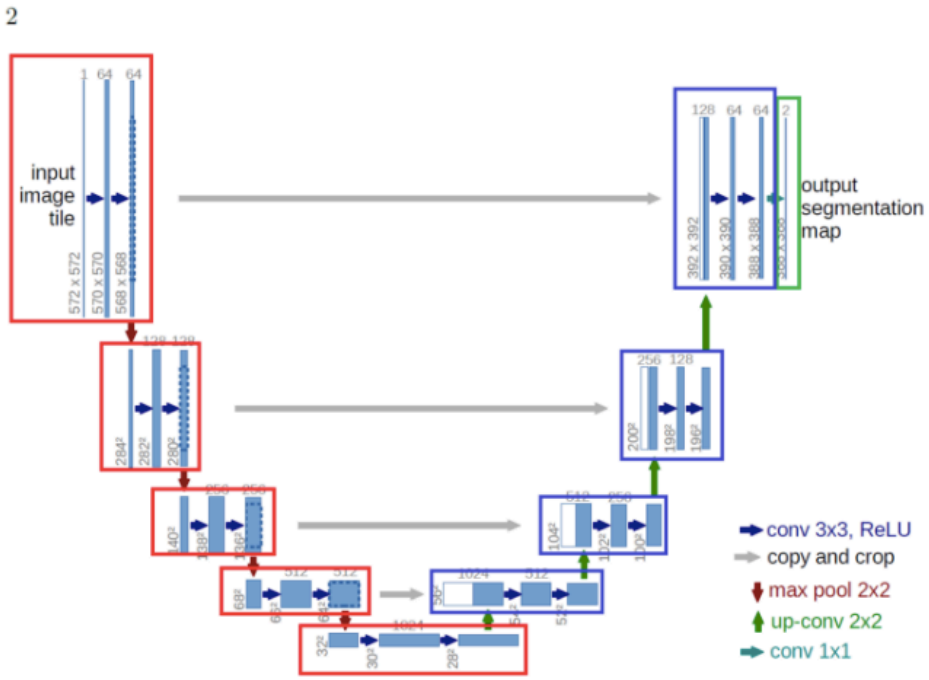


Fig. 1. U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.

Red Box → Representing the left side of U Net
Blue Box → Representing the Right side of U Net
Green Box → Final Bottle neck layer.

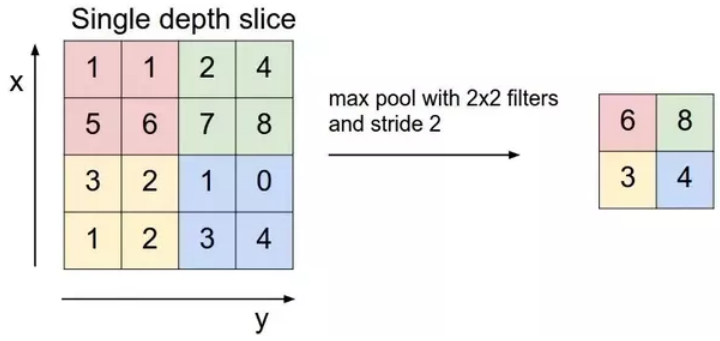
The **U-Net** is a convolutional neural network that was developed for biomedical image segmentation at the Computer Science Department of the University of Freiburg, Germany.The network is based on the fully convolutional network and its architecture was modified and extended to work with fewer training images and to yield more precise segmentations.

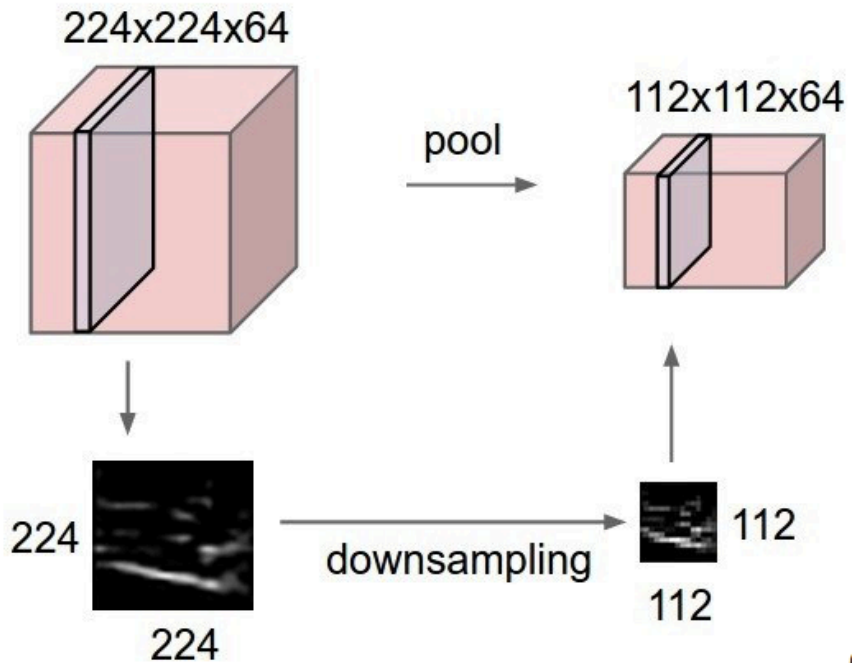
The network consists of a contracting path and an expansive path, which gives it the u-shaped architecture. The contracting path is a typical convolutional network that consists of repeated application of convolutions, each followed by a rectified linear unit (ReLU) and a max pooling

operation. During the contraction, the spatial information is reduced while feature information is increased. The expansive pathway combines the feature and spatial information through a sequence of up-convolutions and concatenations with high-resolution features from the contracting path.

Max-pooling:

Max pooling is a **sample-based discretization process**. The objective is to down-sample an input representation (image, hidden-layer output matrix, etc.), reducing its dimensionality and allowing for assumptions to be made about features contained in the sub-regions binned.
This is done to in part to help over-fitting by providing an abstracted form of the representation. As well, it reduces the computational cost by reducing the number of parameters to learn and provides basic translation invariance to the internal representation.
Max pooling is done by applying a *max filter* to (usually) non-overlapping subregions of the initial representation.
Let's say we have a 4x4 matrix representing our initial input.
Let's say, as well, that we have a 2x2 filter that we'll run over our input. We'll have a **stride** of 2 (meaning the (dx, dy) for stepping over our input will be (2, 2)) and won't overlap regions.
For each of the regions represented by the filter, we will take the ***max*** of that region and create a new, output matrix where each element is the max of a region in the original input.





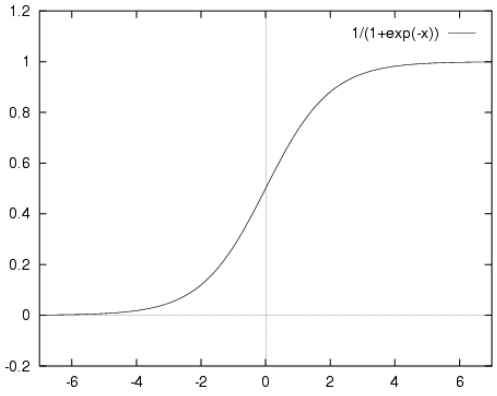
Activation Function:

Activation functions are really important for a Artificial Neural Network to learn and make sense of something really complicated and Non-linear complex functional mappings between the inputs and response variable.They introduce non-linear properties to our Network.**Their main purpose is to convert a input signal of a node in a A-NN to an output signal.** That output signal now is used as a input in the next layer in the stack.

1. Sigmoid Function :

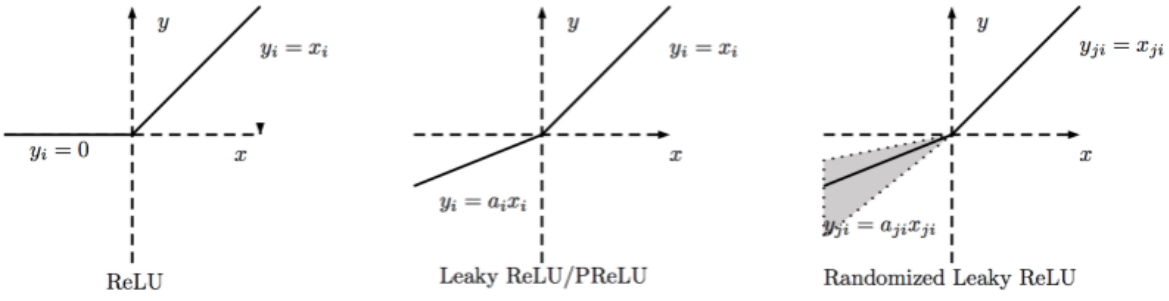
It is a activation function of form **f(x) = 1 / 1 + exp(-x) .** Its Range is between 0 and 1. It is a S—shaped curve. It is easy to understand and apply but it has major reasons which have made it fall out of popularity -

- Vanishing gradient problem
- Secondly , its output isn't zero centered. It makes the gradient updates go too far in different directions. **0 < output < 1, and it makes optimization harder.**
- Sigmoids saturate and kill gradients.
- Sigmoids have slow convergence.



2. ReLu(Rectified Linear Units) function:

It has become very popular in the past couple of years. It was recently proved that it had 6 times improvement in convergence from Tanh function. It's just R(x) = max(0,x) i.e if x < 0 , R(x) = 0 and if x >= 0 , R(x) = x. Hence as seeing the mathematical form of this function we can see that it is very simple and efficient . A lot of times in Machine learning and computer science we notice that most simple and consistent techniques and methods are only preferred and are



best. Hence it avoids and rectifies vanishing gradient problem . Almost all deep learning Models use ReLu nowadays. Three variants of this function are given below.

Convolution Neural Networks(CNN):

In neural networks, Convolutional neural network (ConvNets or CNNs) is one of the main categories to do images recognition, images classifications. Objects detections, recognition faces etc., are some of the areas where CNNs are widely used.
CNN image classifications takes an input image, process it and classify it under certain categories (Eg., Dog, Cat, Tiger, Lion). Computers sees an input image as array of pixels and it depends on the image resolution. Based on the image resolution, it will see h x w x d(h = Height, w = Width, d = Dimension). Eg., An image of 6 x 6 x 3 array of matrix of RGB (3 refers

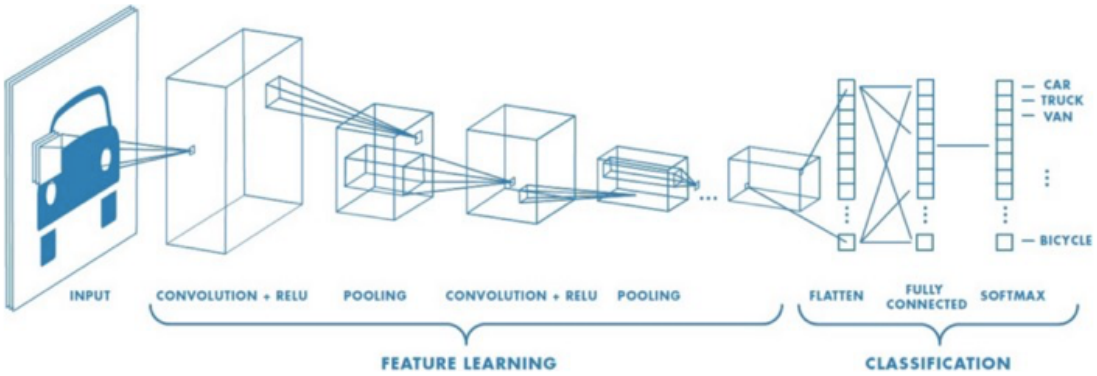


Figure 2 : Neural network with many convolutional layers

to RGB values) and an image of 4 x 4 x 1 array of matrix of grayscale image.

1. Convolution Layer :

- An image matrix (volume) of dimension **(h x w x d)**
- A filter **(f_h x f_w x d)**
- Outputs a volume dimension **(h - f_h + 1) x (w - f_w + 1) x 1**

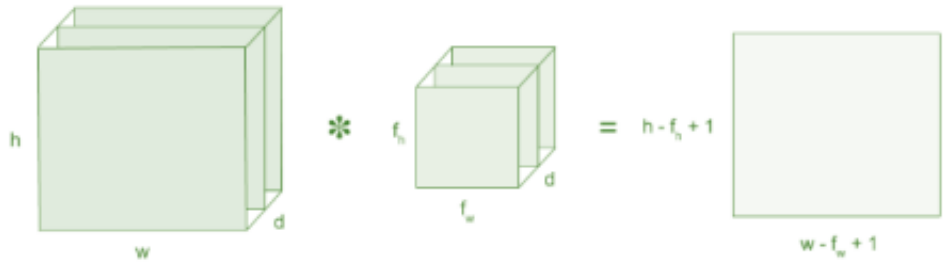


Figure 3: Image matrix multiplies kernel or filter matrix

Convolution is the first layer to extract features from an input image. Convolution preserves the relationship between pixels by learning image features using small squares of input data. It is a mathematical operation that takes two inputs such as image matrix and a filter or kernel.

2. Stride:

Stride is the number of pixels shifts over the input matrix. When the stride is 1 then we move the filters to 1 pixel at a time. When the stride is 2 then we move the filters to 2 pixels at a time and so on. The below figure shows convolution would work with a stride of 2.

3 . Padding :

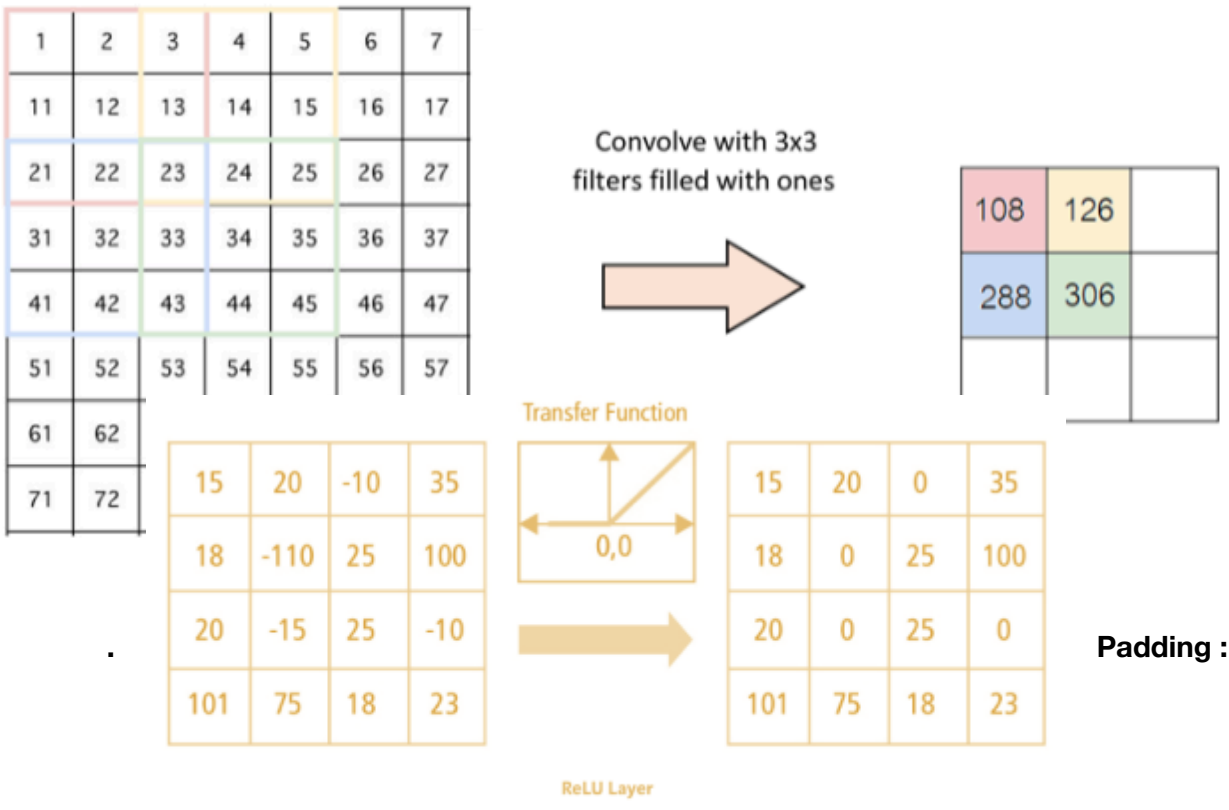


Figure 7 : ReLU operation

Sometimes filter does not fit perfectly fit the input image. We have two options:

- Pad the picture with zeros (zero-padding) so that it fits
- Drop the part of the image where the filter did not fit. This is called valid padding which keeps only valid part of the image.

4. Non Linearity(ReLu):

ReLU stands for Rectified Linear Unit for a non-linear operation. The output is ***ƒ(x) = max(0,x).*** Why ReLU is important : ReLU's purpose is to introduce non-linearity in our ConvNet. Since, the real world data would want our ConvNet to learn would be non-negative linear values.

5. Pooling Layers:

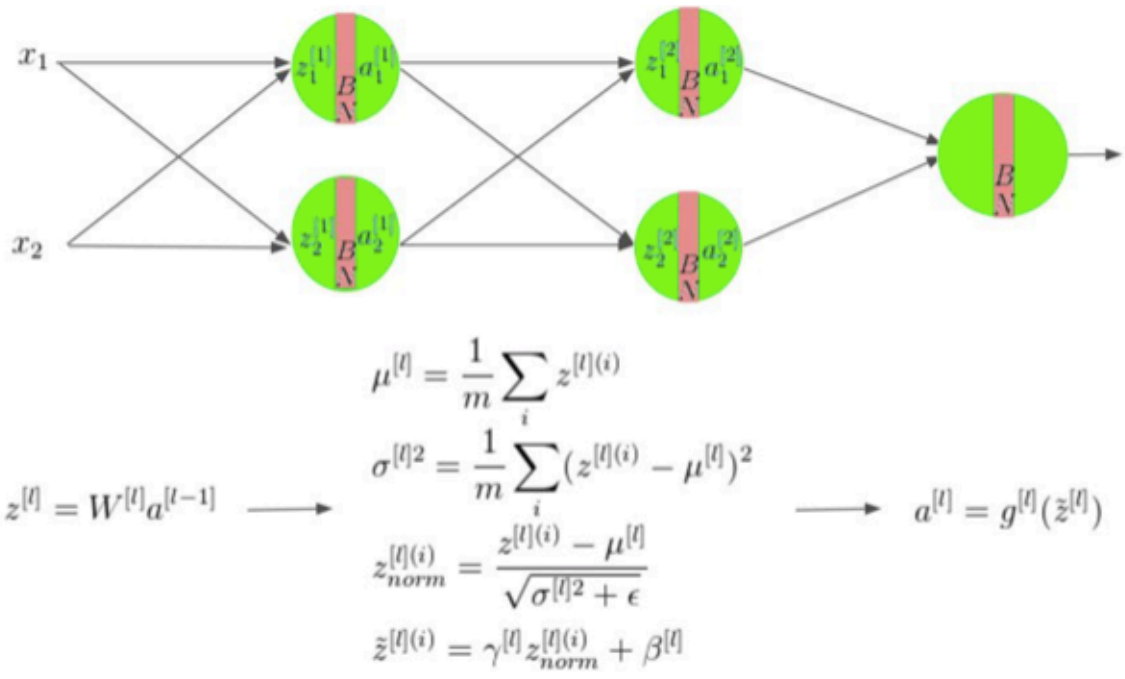


Figure 4: Batch Normalization on a simple network.

Pooling layers section would reduce the number of parameters when the images are too large. Spatial pooling also called subsampling or downsampling which reduces the dimensionality of each map but retains the important information. Spatial pooling can be of different types:

- Max Pooling
- Average Pooling
- Sum Pooling

Max pooling take the largest element from the rectified feature map. Taking the largest element could also take the average pooling. Sum of all elements in the feature map call as sum pooling.

Batch Normalization:

Batch normalization is a technique for improving the performance and stability of artificial neural networks. It is a technique to provide any layer in a neural network with inputs that are zero mean/unit variance.[1] Batch normalization was introduced in a 2015 paper.[2][3] It is used to normalize the input layer by adjusting and scaling the activations.[4]

Batch Normalization in a neural network:

Batch normalization is done individually at each unit. Figure 4 shows how it works on a simple network with input features x_1 and x_2. Let's go over the equations.

In Figure 4, the superscript (i) corresponds to the i^{th} data in the mini batch, the superscript $[l]$ indicates the l^{th} layer in the network, and the subscript k indicates the k^{th} dimension in a given layer in the network. In some places, either of the superscripts or the subscript has been dropped to keep the notations simple.

Batch Normalization is done individually at every hidden unit. Traditionally, the input to a layer $a^{[l-1]}$ goes through an affine transform which is then passed through a non-linearity $g^{[l]}$ such as ReLU or sigmoid to get the final activation a^l from the unit. So,
$a^l = g^{[l]}(W^{[l]}a^{[l-1]} + b^{[l]})$.

But when Batch Normalization is used with a transform BN, it becomes
$a^l = g^{[l]}(BN(W^{[l]}a^{[l-1]}))$

The bias b could now be ignored because its effect is subsumed with the shift parameter β.
The four equations shown in Figure 4 do the following.

1. Calculate mean (μ) of the minibatch.
2. Calculate variance (σ^2) of the minibatch.
3. Calculate z_{norm} by subtracting mean from z and subsequently dividing by standard deviation (σ). A small number, epsilon (ϵ), is added to the denominator to prevent divide by zero.
4. Calculate $\tilde{z}_{norm}$ by multiplying z_{norm} with a scale (γ) and adding a shift (β) and use $\tilde{z}_{norm}$ in place of z as the non-linearity's (e.g. ReLU's) input. The two parameters β and γ are learned during the training process with the weight parameters W.

Note: Batch normalization adds only two extra parameters for each unit. So the representation power of the network is still preserved. If β is set to μ and γ to $\sqrt{\sigma^2 + \epsilon}$, then $\tilde{z}_{norm}$ equals z, thus working as an identity function. Thus introducing batch normalization alone would not reduce the accuracy because the optimizer still has the option to select no normalization effect using the identity function and it would be used by the optimizer to only improve the results.

Dropouts:

Dropout is a regularization technique for neural network models proposed by Srivastava, et al. in their 2014 paper Dropout: A Simple Way to Prevent Neural Networks from Overfitting (download the PDF).

Dropout is a technique where randomly selected neurons are ignored during training. They are "dropped-out" randomly. This means that their contribution to the activation of downstream neurons is temporally removed on the forward pass and any weight updates are not applied to the neuron on the backward pass.

As a neural network learns, neuron weights settle into their context within the network. Weights of neurons are tuned for specific features providing some specialization. Neighboring neurons become to rely on this specialization, which if taken too far can result in a fragile model too specialized to the training data. This reliant on context for a neuron during training is referred to complex co-adaptations.

We can imagine that if neurons are randomly dropped out of the network during training, that other neurons will have to step in and handle the representation required to make predictions for the missing neurons. This is believed to result in multiple independent internal representations being learned by the network.

The effect is that the network becomes less sensitive to the specific weights of neurons. This in turn results in a network that is capable of better generalization and is less likely to overfit the training data.

$$v_{dW} = \beta_1 v_{dW} + (1 - \beta_1)\frac{\partial \mathcal{J}}{\partial W}$$

$$s_{dW} = \beta_2 s_{dW} + (1 - \beta_2)(\frac{\partial \mathcal{J}}{\partial W})^2$$

$$v_{dW}^{corrected} = \frac{v_{dW}}{1 - (\beta_1)^t}$$

$$s_{dW}^{corrected} = \frac{s_{dW}}{1 - (\beta_1)^t}$$

$$W = W - \alpha \frac{v_{dW}^{corrected}}{\sqrt{s_{dW}^{corrected}} + \varepsilon}$$

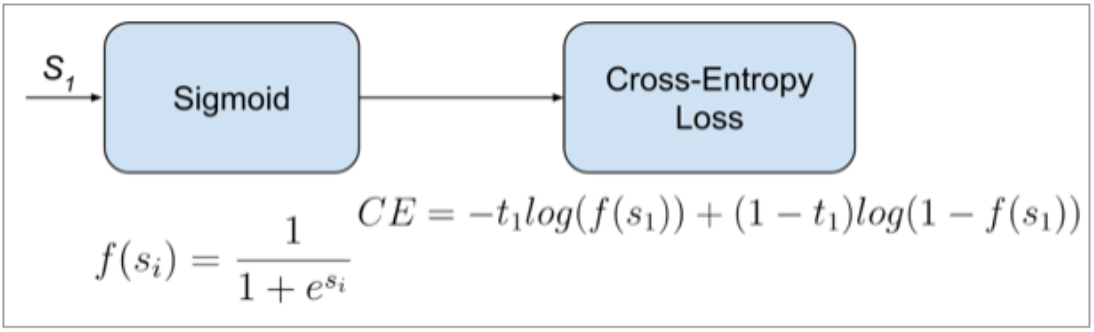
Note

- v_{dW} - the exponentially weighted average of past gradients
- s_{dW} - the exponentially weighted average of past squares of gradients
- β_1 - hyperparameter to be tuned
- β_2 - hyperparameter to be tuned
- $\frac{\partial \mathcal{J}}{\partial W}$ - cost gradient with respect to current layer
- W - the weight matrix (parameter to be updated)
- α - the learning rate
- ϵ - very small value to avoid dividing by zero

Transposed Convolution Layers(Deconvolutions):

Some sources use the name deconvolution, which is inappropriate because it's not a deconvolution. To make things worse deconvolutions do exists, but they're not common in the field of deep learning. An actual deconvolution reverts the process of a convolution. Imagine inputting an image into a single convolutional layer. Now take the output, throw it into a black box and out comes your original image again. This black box does a deconvolution. It is the mathematical inverse of what a convolutional layer does.
A transposed convolution is somewhat similar because it produces the same spatial resolution a hypothetical deconvolutional layer would. However, the actual mathematical operation that's

$$CE = -\sum_{i=1}^{C'=2} t_i log(f(s_i)) = -t_1 log(f(s_1)) + (1 - t_1) log(1 - f(s_1))$$



This would be the pipeline for each one of the C clases. We set C independent binary classification problems $(C' = 2)$. Then we sum up the loss over the different binary problems. s_1 and t_1 are the score and the gorundtruth label for the class C_1, which is also the class C_i in C. $s_2 = 1 - s_1$ and $t_2 = 1 - t_1$ are the score and the groundtruth label of the class C_2, which is not a "class" in our original problem with C classes, but a class we create to set up the binary problem with $C_1 = C_i$. We can understand it as a background class.

The loss can be expressed as:

$$CE = \begin{cases} -log(s_1) & if \quad t_1 = 1 \\ -log(1 - s_1) & if \quad t_1 = 0 \end{cases}$$

Where $t_1 = 1$ means that the class $C_1 = C_i$ is positive for this sample.

In this case, the activation function does not depend in scores of other classes in C more than $C_1 = C_i$. So the gradient respect to the each score s_i in s will only depend on the loss given by its binary problem.

The gradient respect to the score $s_i = s_1$ can be written as:

$$CE = -\sum_{i=1}^{C'=2} t_i log(f(s_i)) = -t_1 log(f(s_1)) + (1 - t_1) log(1 - f(s_1))$$

Where $f()$ is the **sigmoid** function. It can also be written as:

$$\frac{\partial}{\partial s_i} (CE(f(s_i)) = \begin{Bmatrix} s_i - 1 & if \quad t_i = 1 \\ s_i & if \quad t_i = 0 \end{Bmatrix}$$

being performed on the values is different. A transposed convolutional layer carries out a regular convolution but reverts its spatial transformation.

Optimizers:

Adam :

Adaptive Moment Estimation (Adam) combines ideas from both RMSProp and Momentum. It computes adaptive learning rates for each parameter and works as follows.

- First, it computes the exponentially weighted average of past gradients (v_{dW}).
- Second, it computes the exponentially weighted average of the squares of past gradients (s_{dW}).
- Third, these averages have a bias towards zero and to counteract this a bias correction is applied ($v_{dW}^{corrected}$, $s_{dW}^{corrected}$).
- Lastly, the parameters are updated using the information from the calculated averages.

Binary Cross Entropy Loss:

Also called Sigmoid Cross-Entropy loss. It is a Sigmoid activation plus a Cross-Entropy loss. Unlike Softmax loss it is independent for each vector component (class), meaning that the loss computed for every vector component is not affected by other component values. That's why it is used for multi-label classification, were the insight of an element belonging to a certain class should not influence the decision for another class. It's called Binary Cross-Entropy Loss because it sets up a binary classification problem between **C' = 2** classes for every class in **C** as explained above. So when using this Loss, the formulation of **Cross Entropy Loss** for binary problems is often used:

Data(Image) Augmentation:

To build a powerful image classifier using very little training data, image augmentation is usually required to boost the performance of deep networks. **Image augmentation** artificially creates training images through different ways of processing or combination of multiple processing, such as random rotation, shifts, shear and flips, etc.

Lovász-loss:

A tractable surrogate for the optimization of the intersection-over-union measure in neural networks. Reference: https://arxiv.org/abs/1705.08790

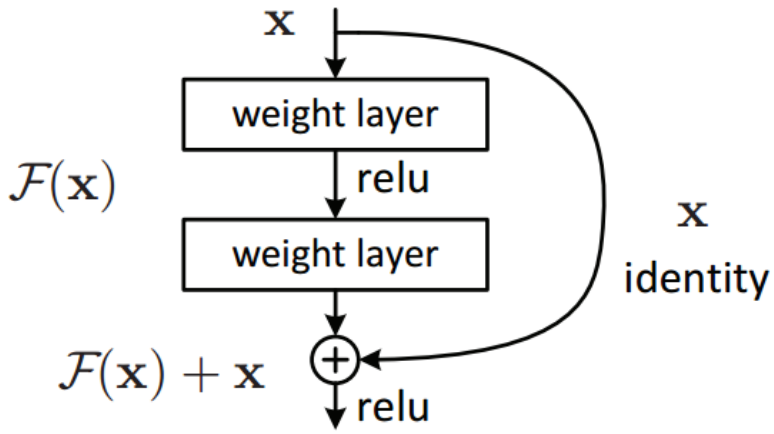
Algorithm 1 Gradient of the Jaccard loss extension $\overline{\Delta_{J_c}}$

Inputs: vector of errors $\boldsymbol{m}(c) \in \mathbb{R}_+^p$
class foreground pixels $\boldsymbol{\delta} = \{\boldsymbol{y}^* = c\} \in \{0, 1\}^p$
Output: $\boldsymbol{g}(\boldsymbol{m})$ gradient of $\overline{\Delta_{J_c}}$ (Equation (9))

$\boldsymbol{\pi} \leftarrow$ decreasing sort permutation for $\boldsymbol{m}$
$\boldsymbol{\delta_\pi} \leftarrow (\delta_{\pi_i})_{i \in [1,p]}$
intersection $\leftarrow \text{sum}(\boldsymbol{\delta}) - \textbf{cumulative\_sum}(\boldsymbol{\delta_\pi})$
union $\leftarrow \text{sum}(\boldsymbol{\delta}) + \textbf{cumulative\_sum}(1 - \boldsymbol{\delta_\pi})$
$\boldsymbol{g} \leftarrow 1 -$ **intersection**$/$**union**
if $p > 1$ **then**
  $\boldsymbol{g}[2:p] \leftarrow \boldsymbol{g}[2:p] - \boldsymbol{g}[1:p-1]$
end if
return $\boldsymbol{g}_{\boldsymbol{\pi}^{-1}}$

Resudial Blocks(Building blocks of ResNet):

In traditional neural networks, each layer feeds into the next layer. In a network with residual blocks, each layer feeds into the next layer and directly into the layers about 2–3 hops away.



Single Residual Block

Benchmark:

The model with the Public Leaderboard score of 0.231 will be used as a benchmark model. Link to the kernel : https://www.kaggle.com/christofhenkel/keras-baseline

This is just a basic CNN model trained with the dataset from the competition.

III. Methodology:

Data Preprocessing:

Upsampling and downsampling is done before training the data and after prediction respectively.

```
def upsample(img):
    if img_size_ori == img_size_target:
        return img
    return resize(img, (img_size_target, img_size_target), mode='constant', preserve_range=True)
    #res = np.zeros((img_size_target, img_size_target), dtype=img.dtype)
    #res[:img_size_ori, :img_size_ori] = img
    #return res

def downsample(img):
    if img_size_ori == img_size_target:
        return img
    return resize(img, (img_size_ori, img_size_ori), mode='constant', preserve_range=True)
    #return img[:img_size_ori, :img_size_ori]
```

Scaling is performed for achieving better accuracy.

In the second u-net model we applied data augmentation for increasing the dataset and thereby accuracy.

```
#Data augmentation
x_train = np.append(x_train, [np.fliplr(x) for x in x_train], axis=0)
y_train = np.append(y_train, [np.fliplr(x) for x in y_train], axis=0)
print(x_train.shape)
print(y_valid.shape)
```

```
(6400, 101, 101, 1)
(800, 101, 101, 1)
```

Reading the image and masks:

Here we load the images and masks into a dataframe and divide it by 255(since it increases the accuracy).

```
[ ]
    train_df["images"] = [np.array(load_img("data/train/images/{}.png".format(idx), color_mode="grayscale")) / 255 for idx in (train_df.index)]
```

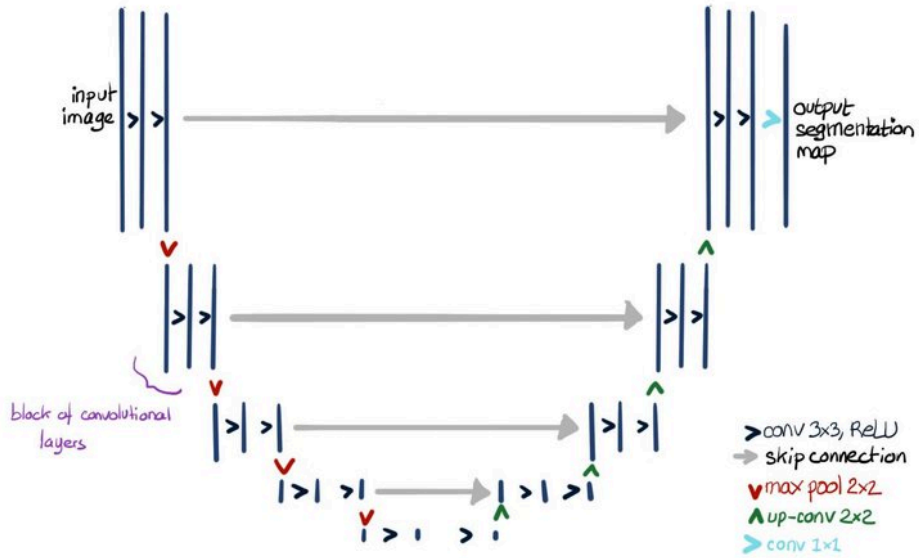
```
[ ]  train_df["masks"] = [np.array(load_img("data/train/masks/{}.png".format(idx), color_mode="grayscale")) / 255 for idx in (train_df.index)]
```

Splitting the dataset into train and test:

```
ids_train, ids_valid, x_train, x_valid, y_train, y_valid, cov_train, cov_test, depth_train, depth_test = train_test_split(
    train_df.index.values,
    np.array(train_df.images.map(upsample).tolist()).reshape(-1, img_size_target, img_size_target, 1),
    np.array(train_df.masks.map(upsample).tolist()).reshape(-1, img_size_target, img_size_target, 1),
    train_df.coverage.values,
    train_df.z.values,
    test_size=0.2, stratify=train_df.coverage_class, random_state=1234)
```

Implementation:

Model 1



We first tried implanting a simple u-net model with conv2d layers, maxpooling, dropouts and 'ReLu' as activation function. The output layer consists of a sigmoid function. Let's call this model as "Model 1".

Code of the u-net architecture(Model 1):

```
def build_model(input_layer, start_neurons):
    # 128 -> 64
    conv1 = Conv2D(start_neurons * 1, (3, 3), activation="relu", padding="same")(input_layer)
    conv1 = Conv2D(start_neurons * 1, (3, 3), activation="relu", padding="same")(conv1)
    pool1 = MaxPooling2D((2, 2))(conv1)
    pool1 = Dropout(0.25)(pool1)

    # 64 -> 32
    conv2 = Conv2D(start_neurons * 2, (3, 3), activation="relu", padding="same")(pool1)
    conv2 = Conv2D(start_neurons * 2, (3, 3), activation="relu", padding="same")(conv2)
    pool2 = MaxPooling2D((2, 2))(conv2)
    pool2 = Dropout(0.5)(pool2)

    # 32 -> 16
    conv3 = Conv2D(start_neurons * 4, (3, 3), activation="relu", padding="same")(pool2)
    conv3 = Conv2D(start_neurons * 4, (3, 3), activation="relu", padding="same")(conv3)
    pool3 = MaxPooling2D((2, 2))(conv3)
    pool3 = Dropout(0.5)(pool3)

    # 16 -> 8
    conv4 = Conv2D(start_neurons * 8, (3, 3), activation="relu", padding="same")(pool3)
    conv4 = Conv2D(start_neurons * 8, (3, 3), activation="relu", padding="same")(conv4)
    pool4 = MaxPooling2D((2, 2))(conv4)
    pool4 = Dropout(0.5)(pool4)

    # Middle
    convm = Conv2D(start_neurons * 16, (3, 3), activation="relu", padding="same")(pool4)
    convm = Conv2D(start_neurons * 16, (3, 3), activation="relu", padding="same")(convm)

    # 8 -> 16
    deconv4 = Conv2DTranspose(start_neurons * 8, (3, 3), strides=(2, 2), padding="same")(convm)
    uconv4 = concatenate([deconv4, conv4])
    uconv4 = Dropout(0.5)(uconv4)
    uconv4 = Conv2D(start_neurons * 8, (3, 3), activation="relu", padding="same")(uconv4)
    uconv4 = Conv2D(start_neurons * 8, (3, 3), activation="relu", padding="same")(uconv4)

    # 16 -> 32
    deconv3 = Conv2DTranspose(start_neurons * 4, (3, 3), strides=(2, 2), padding="same")(uconv4)
    uconv3 = concatenate([deconv3, conv3])
    uconv3 = Dropout(0.5)(uconv3)
    uconv3 = Conv2D(start_neurons * 4, (3, 3), activation="relu", padding="same")(uconv3)
    uconv3 = Conv2D(start_neurons * 4, (3, 3), activation="relu", padding="same")(uconv3)

    # 32 -> 64
    deconv2 = Conv2DTranspose(start_neurons * 2, (3, 3), strides=(2, 2), padding="same")(uconv3)
    uconv2 = concatenate([deconv2, conv2])
    uconv2 = Dropout(0.5)(uconv2)
    uconv2 = Conv2D(start_neurons * 2, (3, 3), activation="relu", padding="same")(uconv2)
    uconv2 = Conv2D(start_neurons * 2, (3, 3), activation="relu", padding="same")(uconv2)

    # 64 -> 128
    deconv1 = Conv2DTranspose(start_neurons * 1, (3, 3), strides=(2, 2), padding="same")(uconv2)
```

```
    uconv1 = concatenate([deconv1, conv1])
    uconv1 = Dropout(0.5)(uconv1)
    uconv1 = Conv2D(start_neurons * 1, (3, 3), activation="relu", padding="same")(uconv1)
    uconv1 = Conv2D(start_neurons * 1, (3, 3), activation="relu", padding="same")(uconv1)

    #uconv1 = Dropout(0.5)(uconv1)
    output_layer = Conv2D(1, (1,1), padding="same", activation="sigmoid")(uconv1)

    return output_layer
```

The build_model() represents our implementation of the u-net model with two sides left and right.

The left side of our u-net consists of 5 blocks(128->64, 64->32, 32->16, 16->8, middle). The first four of these blocks comprises of 2 convolution layers followed by max-pooling and dropout. In the middle block there are two convolution layers.

The right side consists of 5 blocks(8->16, 16->32, 32->64, 64->128, output). The first four consists of 1 deconvolution layer followed by concatenation, dropout and 2 convolution layers. The output layer comprises of a convolution layer with sigmoid activation function.

We compile the model using loss function as binary cross entropy, adam as optimizer and metrics as accuracy.

We trained the model initially with 100 epochs and 32 as batch size but didn't result in a good accuracy.

So next we tried with 200 epochs and 32 batch size. This increased the accuracy.

After this we tried building another model ignorer to achieve more accuracy. Let's name this model as "Model 2".

Model 2 architecture and implementation :

The model 2 is a u-net model with simple ResNet blocks.

a) The network is composed of Convolution Operation, Max Pooling, ReLU Activation, Concatenation and Up Sampling Layers.

b) It consists of a contracting path (left side) and an expansive path (right side).

c) The contracting path follows the typical architecture of a convolutional network. It consists of the repeated application of two 3x3 convolutions (unpadded convolutions), each followed by a rectified linear unit (ReLU) and a 2x2 max pooling operation with stride 2 for downsampling.

d) At each downsampling step we double the number of feature channels.

e) Every step in the expansive path consists of an upsampling of the feature map followed by a 2x2 convolution ("up-convolution") that halves the number of feature channels, a concatenation with the correspondingly cropped feature map from the contracting path, and two 3x3 convolutions, each followed by a ReLU.

f) Up-sampling can be done by using Transpose Convolution Operation.

g) The cropping is necessary due to the loss of border pixels in every convolution.

h) At the final layer a 1x1 convolution is used to map each 64 component feature vector to the desired number of classes.

Some helper methods:

```
def BatchActivate(x):
    x = BatchNormalization()(x)
    x = Activation('relu')(x)
    return x

def convolution_block(x, filters, size, strides=(1,1), padding='same', activation=True):
    x = Conv2D(filters, size, strides=strides, padding=padding)(x)
    if activation == True:
        x = BatchActivate(x)
    return x

def residual_block(blockInput, num_filters=16, batch_activate = False):
    x = BatchActivate(blockInput)
    x = convolution_block(x, num_filters, (3,3) )
    x = convolution_block(x, num_filters, (3,3), activation=False)
    x = Add()([x, blockInput])
    if batch_activate:
        x = BatchActivate(x)
    return x
```

Model 2 code:

```
# Build model
def build_model(input_layer, start_neurons, DropoutRatio = 0.5):
    # 101 -> 50
    conv1 = Conv2D(start_neurons * 1, (3, 3), activation=None, padding="same")(input_layer)
    conv1 = residual_block(conv1,start_neurons * 1)
    conv1 = residual_block(conv1,start_neurons * 1, True)
    pool1 = MaxPooling2D((2, 2))(conv1)
    pool1 = Dropout(DropoutRatio/2)(pool1)

    # 50 -> 25
    conv2 = Conv2D(start_neurons * 2, (3, 3), activation=None, padding="same")(pool1)
    conv2 = residual_block(conv2,start_neurons * 2)
    conv2 = residual_block(conv2,start_neurons * 2, True)
    pool2 = MaxPooling2D((2, 2))(conv2)
    pool2 = Dropout(DropoutRatio)(pool2)

    # 25 -> 12
    conv3 = Conv2D(start_neurons * 4, (3, 3), activation=None, padding="same")(pool2)
    conv3 = residual_block(conv3,start_neurons * 4)
    conv3 = residual_block(conv3,start_neurons * 4, True)
    pool3 = MaxPooling2D((2, 2))(conv3)
    pool3 = Dropout(DropoutRatio)(pool3)

    # 12 -> 6
    conv4 = Conv2D(start_neurons * 8, (3, 3), activation=None, padding="same")(pool3)
    conv4 = residual_block(conv4,start_neurons * 8)
    conv4 = residual_block(conv4,start_neurons * 8, True)
    pool4 = MaxPooling2D((2, 2))(conv4)
    pool4 = Dropout(DropoutRatio)(pool4)
```

```
    # Middle
    convm = Conv2D(start_neurons * 16, (3, 3), activation=None, padding="same")(pool4)
    convm = residual_block(convm,start_neurons * 16)
    convm = residual_block(convm,start_neurons * 16, True)

    # 6 -> 12
    deconv4 = Conv2DTranspose(start_neurons * 8, (3, 3), strides=(2, 2), padding="same")(convm)
    uconv4 = concatenate([deconv4, conv4])
    uconv4 = Dropout(DropoutRatio)(uconv4)

    uconv4 = Conv2D(start_neurons * 8, (3, 3), activation=None, padding="same")(uconv4)
    uconv4 = residual_block(uconv4,start_neurons * 8)
    uconv4 = residual_block(uconv4,start_neurons * 8, True)

    # 12 -> 25
    #deconv3 = Conv2DTranspose(start_neurons * 4, (3, 3), strides=(2, 2), padding="same")(uconv4)
    deconv3 = Conv2DTranspose(start_neurons * 4, (3, 3), strides=(2, 2), padding="valid")(uconv4)
    uconv3 = concatenate([deconv3, conv3])
    uconv3 = Dropout(DropoutRatio)(uconv3)

    uconv3 = Conv2D(start_neurons * 4, (3, 3), activation=None, padding="same")(uconv3)
    uconv3 = residual_block(uconv3,start_neurons * 4)
    uconv3 = residual_block(uconv3,start_neurons * 4, True)

    # 25 -> 50
    deconv2 = Conv2DTranspose(start_neurons * 2, (3, 3), strides=(2, 2), padding="same")(uconv3)
    uconv2 = concatenate([deconv2, conv2])

    uconv2 = Dropout(DropoutRatio)(uconv2)
    uconv2 = Conv2D(start_neurons * 2, (3, 3), activation=None, padding="same")(uconv2)
    uconv2 = residual_block(uconv2,start_neurons * 2)
    uconv2 = residual_block(uconv2,start_neurons * 2, True)

    # 50 -> 101
    #deconv1 = Conv2DTranspose(start_neurons * 1, (3, 3), strides=(2, 2), padding="same")(uconv2)
    deconv1 = Conv2DTranspose(start_neurons * 1, (3, 3), strides=(2, 2), padding="valid")(uconv2)
    uconv1 = concatenate([deconv1, conv1])

    uconv1 = Dropout(DropoutRatio)(uconv1)
    uconv1 = Conv2D(start_neurons * 1, (3, 3), activation=None, padding="same")(uconv1)
    uconv1 = residual_block(uconv1,start_neurons * 1)
    uconv1 = residual_block(uconv1,start_neurons * 1, True)

    #uconv1 = Dropout(DropoutRatio/2)(uconv1)
    #output_layer = Conv2D(1, (1,1), padding="same", activation="sigmoid")(uconv1)
    output_layer_noActi = Conv2D(1, (1,1), padding="same", activation=None)(uconv1)
    output_layer =  Activation('sigmoid')(output_layer_noActi)

    return output_layer
```

This is similar to model 1 except that it uses ResNet blocks. The output layer consists of sigmoid activation function.

We compiled the model using binary crossentropy as loss function, adam as the optimizer and my_iou_metric as metrics.

my_iou_metric is implemented as follows:

```
def get_iou_vector(A, B):
    batch_size = A.shape[0]
    metric = []
    for batch in range(batch_size):
        t, p = A[batch]>0, B[batch]>0
        intersection = np.logical_and(t, p)
        union = np.logical_or(t, p)
        iou = (np.sum(intersection > 0) + 1e-10 )/ (np.sum(union > 0) + 1e-10)
        thresholds = np.arange(0.5, 1, 0.05)
        s = []
        for thresh in thresholds:
            s.append(iou > thresh)
        metric.append(np.mean(s))

    return np.mean(metric)

def my_iou_metric(label, pred):
    return tf.py_func(get_iou_vector, [label, pred>0.5], tf.float64)

def my_iou_metric_2(label, pred):
    return tf.py_func(get_iou_vector, [label, pred >0], tf.float64)
```

We created two files to save the model.

```
version = 5
basic_name = f'Unet_resnet_v{version}'
save_model_name = basic_name + '.model'
submission_file = basic_name + '.csv'

print(save_model_name)
print(submission_file)
```

```
Unet_resnet_v5.model
Unet_resnet_v5.csv
```

First we tried training the model with 100 epochs and 32 as the batch size but the my_iou_metric score was almost 0.78.

We again retrained the model with 80 epochs and 64 as the batch size. This resulted in an increase in my_iou_metric score which went above 0.8

For better accuracy we again compiled and trained the model with new loss function i.e lovász function and metric as my_iou_metric_2

```
model1 = load_model(save_model_name,custom_objects={'my_iou_metric': my_iou_metric})
# remove later activation layer and use losvasz loss
input_x = model1.layers[0].input

output_layer = model1.layers[-1].input
model = Model(input_x, output_layer)
c = optimizers.adam(lr = learning_rate)

# lovasz_loss need input range (-∞, +∞), so cancel the last "sigmoid" activation
# Then the default threshod for pixel prediction is 0 instead of 0.5, as in my_iou_metric_2.
model.compile(loss=lovasz_loss, optimizer=c, metrics=[my_iou_metric_2])

model.summary()
```

We trained the model with 180 epochs and 64 as batch size.We got a better my_iou_metric_2 score than the previous one.

Refinement:

While training the Model 2 we again changed epochs to 200 and batch size to 32. This further increased the my_iou_metric_2 score.

```
early_stopping = EarlyStopping(monitor='val_my_iou_metric_2', mode = 'max',patience=20, verbose=1)
model_checkpoint = ModelCheckpoint(save_model_name,monitor='val_my_iou_metric_2',
                                   mode = 'max', save_best_only=True, verbose=1)
reduce_lr = ReduceLROnPlateau(monitor='val_my_iou_metric_2', mode = 'max',factor=0.5, patience=5, min_lr=0.0001, verbose=1)
epochs = 200
batch_size = 32

history = model.fit(x_train, y_train,
                    validation_data=[x_valid, y_valid],
                    epochs=epochs,
                    batch_size=batch_size,
                    callbacks=[ model_checkpoint,reduce_lr,early_stopping],
                    verbose=2)
```

IV. Results:

Model Evaluation and Validation:

After submitting the submission file of **Model 1** to kaggle we got a score of **0.741**

2424	▲ 13	**Rocky**		0.742102	1	4mo
2425	▼ 1	**Alessio Borgheresi**		0.741523	14	5mo
2426	▲ 51	**Bit_Wise**		0.741341	48	2mo
2427	▲ 26	**Amartya Kalapahar**		0.741254	2	3mo
2428	▲ 16	**rmmg**		0.741254	26	3mo

After submitting the submission file of **Model 2** to kaggle we got a score of **0.782**

Your most recent submission

Name	Submitted	Wait time	Execution time	Score
submission_file.csv	a minute ago	1 seconds	12 seconds	0.782783

Complete

Jump to your position on the leaderboard ▾

So we choose the **Model 2** since the score in the leaderboard is more.

Also some notable plotting can be given as follows.

In Model 1:

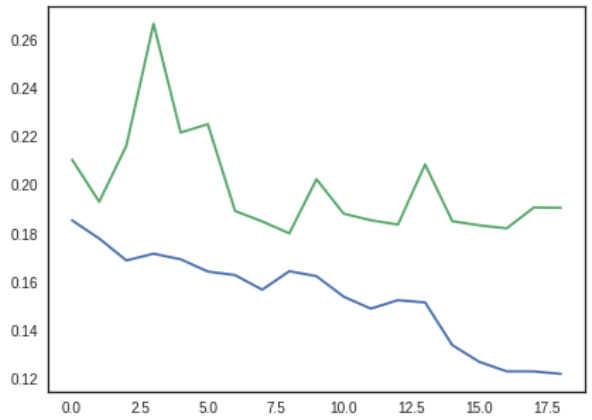


Fig. 1

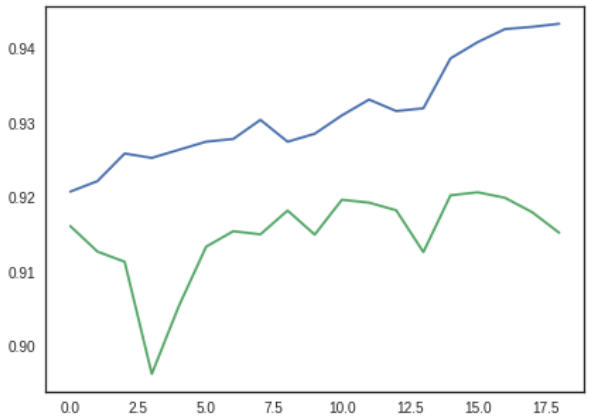


Fig. 2

Fig. 1: Train loss(blue) vs. Validation Loss(Green)

Fig. 2: Train Accuracy vs. Validation Accuracy

In Model 2:

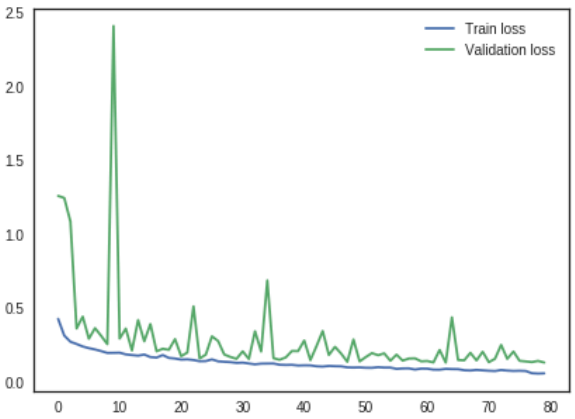


Fig. 1

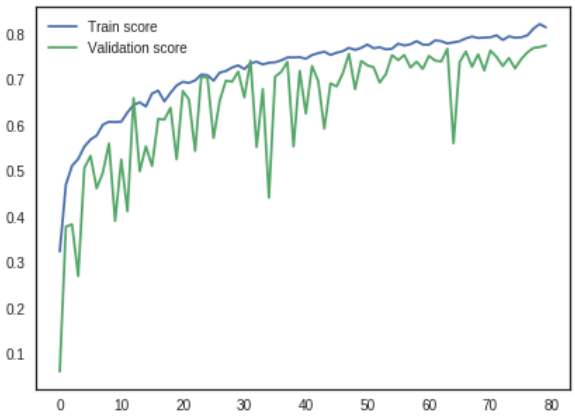


Fig. 2

Fig. 1: Train loss(blue) vs. Validation Loss(Green)

Fig. 2: Train Score vs. Validation Score

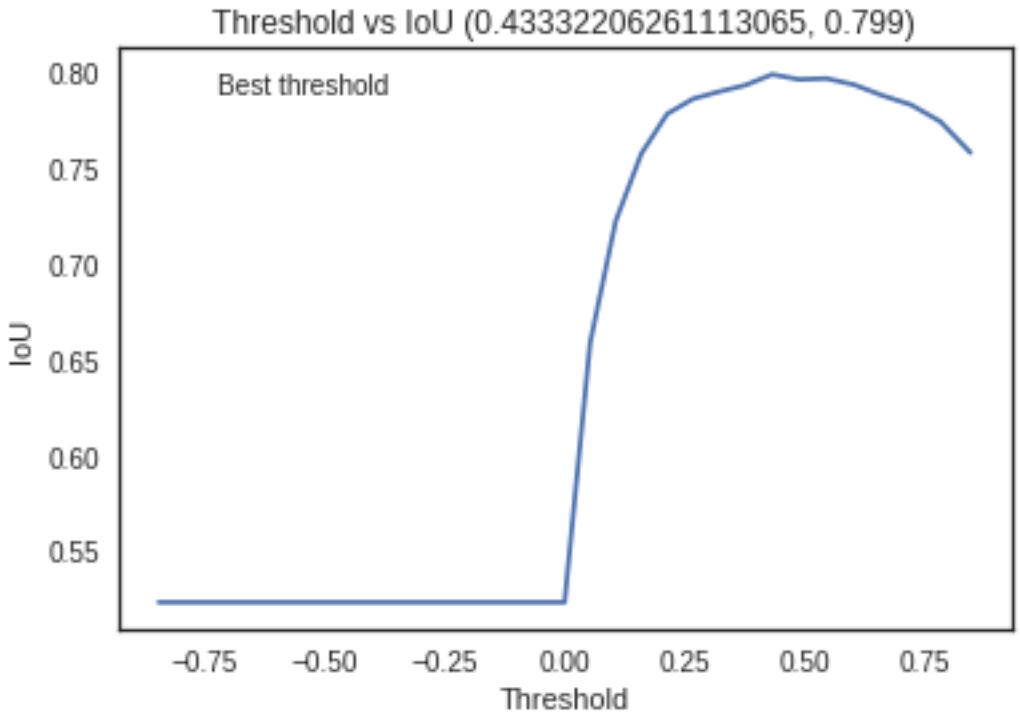
Justification:

Our chosen benchmark model had a kaggle leaderboard score of 0.231 but our final model i.e Model 2 got a leaderboard score of 0.782 which is many times better than the benchmark model.

V. Conclusion:

Free-Form Visualization:

We show a plot of threshold vs IoU here. Instead of using default 0 as threshold, we use validation data to find the best threshold.



Reflection:

I choose this project since it allowed me to tackle new challenges. First of all I didn't know much about seismic data. Then after reading some research papers I got acquainted with seismic data. Then I learnt how to handle these data since previously I dealt with only images in a deep learning problem but here there was image with depth value and mask. The more challenging part was segmentation. This project is purely based on image segmentation. I

found this very interesting and so I started learning more about segmentation by reading alot of research papers. But mostly they were about medical data. This is where I discovered u-net model which was first used for medical image segmentation. I went through various GitHub repositories in order to understand the implementation parts. Naturally, I learnt alot from this and also some kaggle kernels were very helpful for understanding purpose.

One of the difficult part of the project was to design the model with right parameters and proper loss function. Interestingly I learnt about a new loss function i.e lovász loss(softmax and hinge).

This model can be used for practical scenario but since the score is not that high so there can be a bit inefficiency but since salt segmentation is not a critical problem like medical so it can be used.

Improvement:

Our model can be further improved. We can use other encoder such as ResNet34 (pretrained on imagenet dataset). Various other augmentations like brightness, contrast etc. could have been performed ignorer to get a better accuracy. My final model can be used as a benchmark model since the winner of this kaggle challenge has a score of 0.89. Their final model a blend of ResNeXt50 and resnet*34*pad_128.

In resnet34pad_128 they used:
Input: 101 -> pad to 128
Encoder: ResNet34 + scSE (conv7x7 -> conv3x3 and remove first max pooling)
Center Block: Feature Pyramid Attention (remove 7x7)
Decoder: conv3x3, transposed convolution, scSE + hyper columns
Loss: Lovasz

Training overview:
Optimizer: SGD. Batch size: 32.

1. Pretrain on pseudolabels for 150 epochs (50 epochs per cycle with cosine annealing, LR 0.01 -> 0.001)
2. Finetune on train data. 5 folds, 4 snapshots with cosine annealing LR, 50 epochs each, LR 0.01 -> 0.001

In ResNext50 they used:

Input: 101 -> resize to 192 -> pad to 224
Encoder: ResNeXt50 pretrained on ImageNet
Decoder: conv3x3 + BN, Upsampling, scSE
Training overview:
Optimizer: RMSprop. Batch size: 24

1. Loss: BCE+Dice. Reduce LR on plateau starting from 0.0001
2. Loss: Lovasz. Reduce LR on plateau starting from 0.00005
3. Loss: Lovasz. 4 snapshots with cosine annealing LR, 80 epochs each, LR starting from 0.0001