AUTOMATED SEGMENTATION OF BREAST LESIONS USING DEEP NEURAL NETWORK

CONTENTS

CHAPTER **Page No.**

CHAPTER 1

INTRODUCTION

Breast cancer accounts for 22.9% of invasive cancers and 13.7% of cancer-related deaths in women worldwide, according to the International Agency for Research on Cancer[5]. One in eight women is expected to develop invasive breast cancer over the course of her lifetime [7]. Routine mammography is standard for preventive care and detection of breast cancer before biopsy. However, it is still a manual process, prone to human error due to high variability in mass appearance [2] and low signal-to-noise ratio, and thus can cause unnecessary biopsies or worse, missed malignant masses.

In this project, we try to segment and to classify breast lesions from mammography images. To achieve segmentation and mask generation of breast lesions, we use the U-Net architecture [1]. U-Nets train faster and require a lower number of training labeled images, since annotations are difficult.

Convolutional networks are employed to classify the lesion from the U-Net output as either Benign or Malignant. Conv-nets are not only improving for whole-image classification, but also making progress on local tasks with structured output. These include advances in bounding box detection, part and key-point prediction and local correspondence. Convolutional neural networks are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other.

CHAPTER 2

LITERATURE SURVEY

2.1 Segmentation

Convolutional networks are powerful visual models that yield hierarchies of features. The convolutional networks by themselves, train end-to-end, pixels-to-pixels, exceed the state-of-the-art in semantic segmentation. Segmentation is essential for image analysis tasks. Semantic segmentation describes the process of associating each pixel of an image with a class label, (such as flower, person, road, sky, ocean, or car). The networks take inputs of arbitrary size and produce correspondingly-sized output with efficient inference and learning. The convolutional networks, explain their application to spatially dense prediction tasks, and draw connections to prior models. The adaptation of contemporary classification networks (AlexNet, the VGG net and GoogLeNet) into fully convolutional networks and transference of their learned representations by fine-tuning to the segmentation task. Then define a skip architecture that combines semantic information from a deep, coarse layer with appearance information from a shallow, fine layer to produce accurate and detailed segmentations. Our fully convolutional network achieves state-of-the-art segmentation of PASCAL VOC (20% relative improvement to 62.2% mean IU on 2012), NYUDv2, and SIFT Flow, while inference takes less than one fifth of a second for a typical image.

2.2 Classification

Neural network as a fundamental classifier algorithm is widely used in manu image classification issues. With rapid development of high performance computing devices and parallel computing devices, convolutional neural network also draws increasingly more attention from many researchers. A tough image classification problem was solved using neural network classifier that implemented a back-propagation neural network. The functionality of the dropout layer and rectified linear neuron were tested.

The CNN's layers architecture: Convolutional layer, Pooling layer, Fully-connected layer. The two main processes of the CNN algorithm are the convolution and sampling happen on the convolutional and the max pooling layers. Convolutional networks are used for recognition.

2.3 SEGMENTATION APPROACHES

2.3.1 FCN (Fully Convoluted Networks)

Mass Segmentation from Mammograms was carried out by employing a FCN to model a potential function, which in turn was followed by a Conditional Random Field (CRF) [3]. The shape and appearance priori play important roles in mammogram mass segmentation. The distribution of labels varies greatly with position in the mammographic mass segmentation. From observation, most of the masses are located in the center of region of interest (ROI).

In the approach, adversarial training is used to handle the small size of training data to reduce over-fitting and increasing robustness. Accuracies of Adversarial FCN-CRF are 2-3% higher than those Joint FCN-CRF. The adversarial training improves the FCN and Joint FCN-CRF for boundary region segmentation. Due to low contrast of the mammogram, image enhancing techniques are also done. Such pre-processing makes the neural network converge faster. Further augmentation like flipping the image makes the training set 4 times as large as the original training set. The testing was performed on two publicly available datasets DDSM-BCRP and INbreast.

2.3.2 Deep Convolutional Neural Networks

Deep Convolutional Neural Networks for breast lesion segmentation, here every pixel is treated as a possible center of a lesion. After each layer, Deep CNNs represent the data in a more abstract way. They are highly proficient and extremely powerful. A system of DCNN was proposed that out-performed a state-of-the-art Computer Aided Detection (CAD) .A significant advantage because CNN learns from data and does not rely on domain experts, making development easier and faster. CAD systems are also error prone because they are developed in what way a human perceives the problem and

so also tend to make similar errors. The proposed method obtained a AUC of 0.929 with CNN and Augmentation [8]. The AUC can be further extended to 0.941 when Manual Features were also incorporated.

2.3.3 U-Net

U-Net architecture learns from data and does not rely on domain experts, making development easier and faster. The main issue is that there are very few annotated images and sliding window methods take a lot memory space to train. U-Net was brought about to solve this. This method outperforms older techniques like Sliding Window. To supplement a usual contracting network by successive layers, where pooling operators are replaced by up-sampling operators. Hence, these layers increase the resolution of the output. In order to localize, high resolution features from the contracting path are combined with the up-sampled output. Extensive data augmentation is applied if the training data is limited. A warping error of 0.000353 was obtained [1], this performed significantly better than IDSIA, IDSIA-SCI and DIVE. U-Net is mainly used only for segmentation as it does not perform a good job with classification jobs.

2.3.4 Y-Net

Generation of tissue-level segmentation masks for breast cancer diagnosis. This method efficiently segments different types of tissues in breast biopsy images while simultaneously predicting a discriminative map for identifying important areas in an image. Y-Net is an extension of U-Net, by adding a parallel branch for discriminative map generation and by supporting convolutional block modularity, which allows the user to adjust network efficiency without altering the network topology. This system provides a method for Joint Segmentation and Classification. Y-Net adds an additional branch to U-Net architecture that outputs the classification label. Y-Net is fully convolutional and does not have any region proposal network [4]. The Y-Net outperforms the plain and residual encoder-decoder networks by 6%. Fewer learning parameters were required to achieve the same accuracy as state-of-the-art methods.

2.4 CLASSIFICATION

2.4.1 CNN

A Convolution Neural Network (CNN) to study the two-fold applicability of CNN to improve the breast cancer diagnosis. One contribution of this study is to investigate the advantages of recombined images from developed using 49 contrast- enhanced digital mammography (CEDM) in helping the diagnosis of breast lesions using a Deep-CNN method. CEDM is a imaging modality providing information from standard digital mammography (DM) combined with enhancement characteristics related to neoangiogenesis (similar to MRI). A Convolution Neural Network (CNN) to derive “virtual” recombined images from low energy(LE) images, and a CNN is employed to extract novel features to classify the cases as benign vs. malignant. Firstly, a CNN is developed using 49 contrast- enhanced digital mammography(CEDM) cases collected to prove contributions from recombined images for improve breast cancer diagnosis (0.86 in accuracy using LE imaging vs. 0.90 in accuracy using both LE and recombined imaging). Then a CNN using same 49 CEDM cases to learn the nonlinear mapping from LE to recombined images. Using DM alone provides 0.91 in accuracy whereas CNN provides diagnostic accuracy to 0.95.

2.4.2 LeNet

Digitization of hand-written documents was achieved using a multi-layer CNN, this was the first successful approach. Optical Character Recognition (OCR) was performed on the MNIST dataset. The handwritten numbers were fed into the CNN, and training was done in a Gradient based learning procedure. Artificially augmenting the dataset only brought about a minor change in accuracy. LeNet-1 made it clear that a larger CNN was needed to make optimal use of the training dataset. So, LeNet-4 was introduced, it contained more layers and the Test error was only 1.1%. LeNet-5 with Augmentation applied brought the error to 0.7% [6].

CHAPTER 3

LOSS FUNCTION AND HYPERPARAMETER MINING

3.1 BLOCK DIAGRAM

3.2 LOSS FUNCTION

The concept of mean squared error is an important criterion that is utilized in order to measure the performance of an estimator. The mean squared error, abbreviated as MSE, is quite important for relaying the concepts of precision, bias and accuracy during the statistical estimation.

The measure of mean squared error requires a target of prediction or estimation along with a predictor or estimator which is said to be the function of the given data. MSE is defined as the average of squares of the "errors".

Here, the error is said to be the difference between the attribute which is to be estimated and the estimator. The mean squared error may be called a risk function which

corresponds to the expected value of the loss of squared error. This difference or loss could be developed due to the randomness or due to the estimator does not depict the information which could provide a more accurate estimate.

The mean squared error can be referred to the second moment of the error measured about the origin. It incorporates both the variance and bias of the estimator. If an estimator is unbiased estimator, then its mean squared error is same as the variance of the estimator. The unit of MSE is the same as the unit of measurement for the quantity which is being estimated.

Let us suppose that *Xi*^ be the vector denoting values of n number of predictions. Also, *Xi* be a vector representing n number of true values.

Then, the formula for mean squared error is given below:

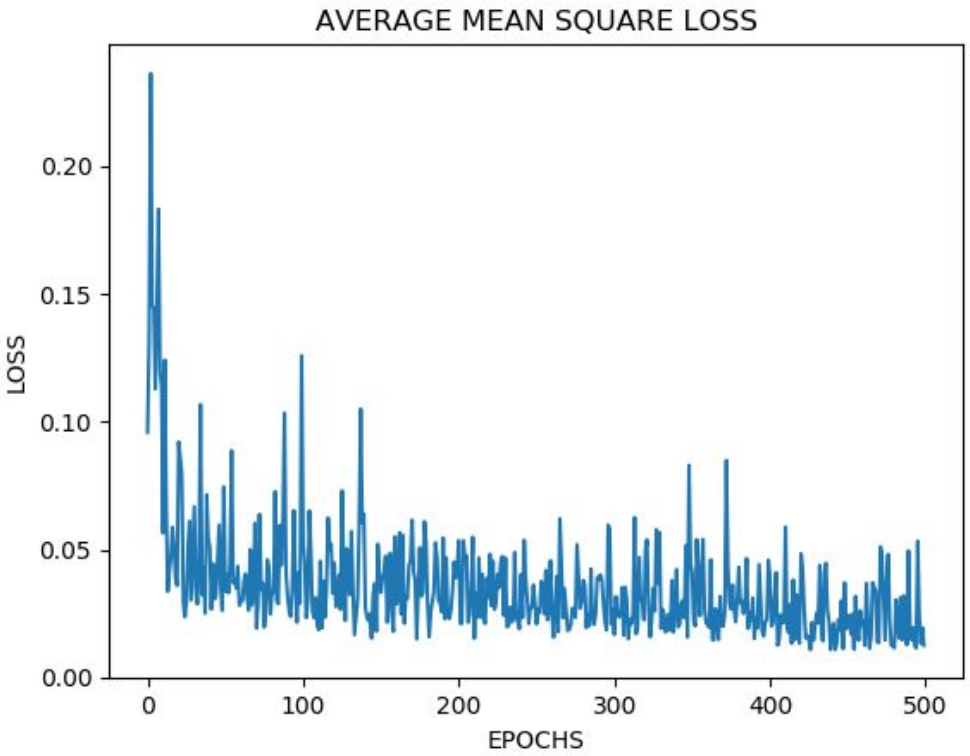
$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (\hat{X}_i - X_i)^2$$

In more general language, if θ be some unknown parameter and $\hat{\theta}$ be the corresponding estimator, then the formula for mean squared error of the given estimator is:

$$\text{MSE}(\hat{\theta}) = E[(\hat{\theta} - \theta)^2]$$

It is to be noted that technically MSE is not a random variable, because it is an expectation. It is subjected to the estimation error for certain given estimator of θ with respect to unknown true value. Therefore, the estimation of mean squared error of an estimated parameter is actually a random variable.

3.2.2 LOSS FUNCTION RESULTS



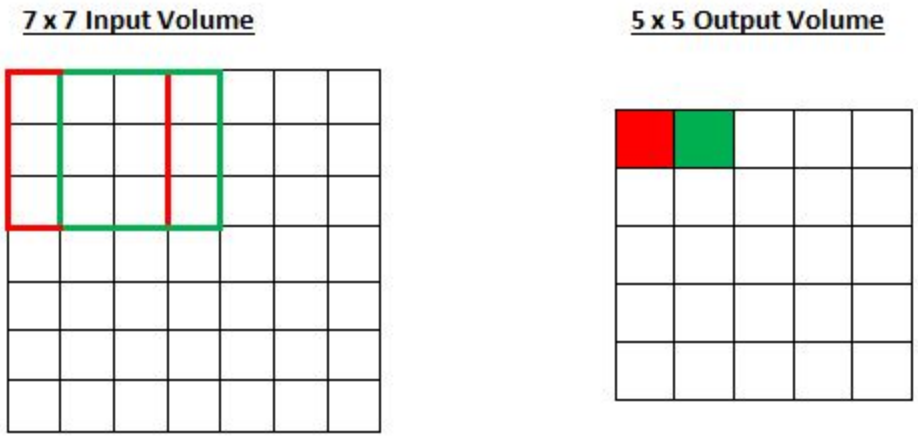
3.3 HYPERPARAMETER MINING

Hyperparameters are the variables which determines the network structure (Eg: Number of Hidden Units) and the variables which determine how the network is trained(Eg: Learning Rate).

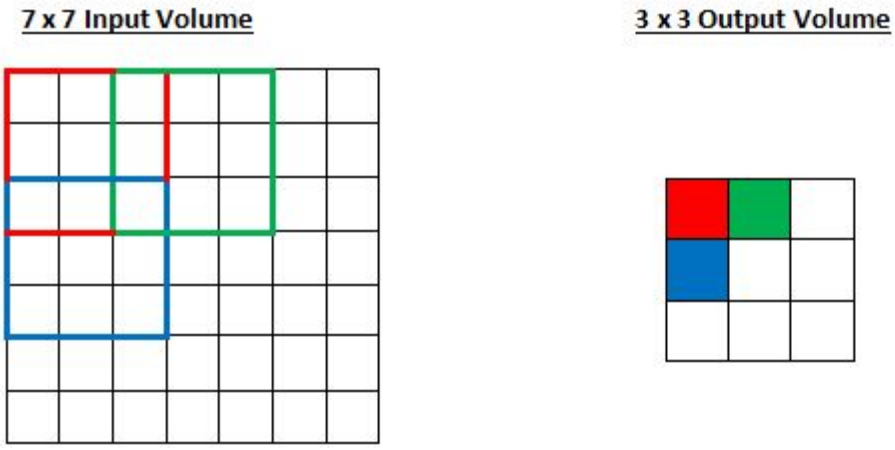
Hyperparameters are set before training before optimizing the weights and bias.

3.3.1 STRIDE

Stride controls how the filter convolves around the input volume. In the example we had in part 1, the filter convolves around the input volume by shifting one unit at a time. The amount by which the filter shifts is the stride. In that case, the stride was implicitly set at 1. Stride is normally set in a way so that the output volume is an integer and not a fraction. Let's look at an example .In a 7 x 7 input volume, a 3 x 3 filter (Disregard the 3rd dimension for simplicity), and a stride of 1. This is the case that we're accustomed to.



When stride increases to 2 .the output volume would be



the receptive field is shifting by 2 units now and the output volume shrinks as well. Notice that if we tried to set our stride to 3, then we'd have issues with spacing and making sure the receptive fields fit on the input volume. Normally, programmers will increase the stride if they want receptive fields to overlap less and if they want smaller spatial dimensions.

3.3.2 ZERO PADDING

Zero-padding refers to the process of symmetrically adding zeroes to the input matrix. It's a commonly used modification that allows the size of the input to be adjusted to our requirement. It is mostly used in designing the CNN layers when the dimensions of the input volume need to be preserved in the output volume.

0	0	0	0	0	0
0	35	19	25	6	0
0	13	22	16	53	0
0	4	3	7	10	0
0	9	8	1	3	0
0	0	0	0	0	0

If you have a stride of 1 and if you set the size of zero padding to

$$Zero\ Padding = \frac{(K-1)}{2}$$

where K is the filter size, then the input and output volume will always have the same spatial dimensions.

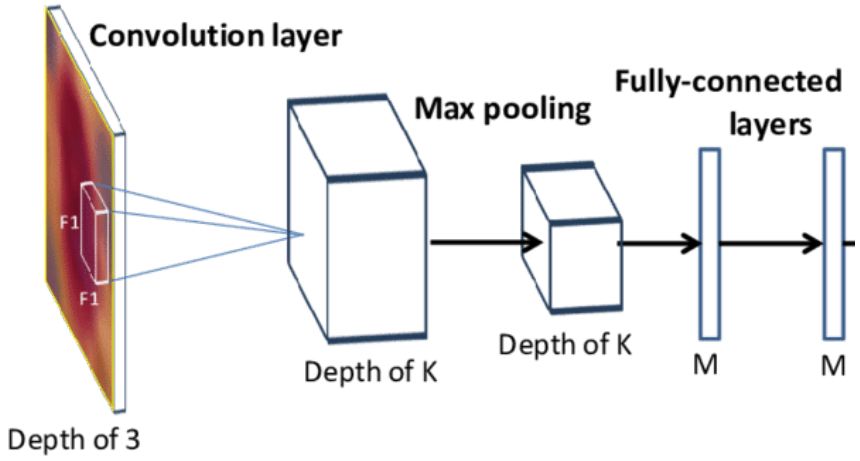
The formula for calculating the output size for any given conv layer is

$$O = \frac{(W-K+2P)}{S} + 1$$

where O is the output height/length, W is the input height/length, K is the filter size, P is the padding, and S is the stride.

3.3.3 DEPTH

The **depth** of the output volume is a hyperparameter: it corresponds to the number of filters we would like to use, each learning to look for something different in the input. For example, if the first Convolutional Layer takes as input the raw image, then different neurons along the depth dimension may activate in presence of various oriented edges, or blobs of color. We will refer to a set of neurons that are all looking at the same region of the input as a **depth column**



3.4 INTERSECTION OVER UNION

Intersection over Union is an evaluation metric used to measure the accuracy of an object detector on a particular dataset. We often see this evaluation metric used in object detection challenges such as the popular PASCAL VOC challenge. We can typically type Intersection Over Union used to evaluate the performance of HOG + Linear SVM object detectors and Convolutional Neural Network detectors (R-CNN , Faster R-CNN , YOLO , etc).

Intersection over Union is simply an evaluation metric. Any algorithm that provides predicted bounding boxes as output can be evaluated using IoU. The ground truth bounding boxes (the hand lebeled bounding boxes from the testing set that specify where in the image our object is).The predicted bounding boxes from our model. As long as we have these two sets of bounding boxes we can apply Intersection Over Union .

CHAPTER 4

SYSTEM ARCHITECTURE

This project makes use of UNET for segmentation and CNN for classification.

4.1 UNET

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.

Compared to FCN-8, the two main differences are (1) U-net is symmetric and (2) the skip connections between the down sampling path and the up sampling path apply a concatenation operator instead of a sum. These skip connections intend to provide local information to the global information while upsampling. Because of its symmetry, the network has a large number of feature maps in the upsampling path, which allows to transfer information. By comparison, the basic FCN architecture only had number of classes feature maps in its upsampling path.

The U-Net owes its name to its symmetric shape, which is different from other FCN variants.

4.1.1 NETWORK ARCHITECTURE

The network architecture is illustrated in Figure 4.1. 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 3x3 convolutions (unpadded convolutions), each followed by a rectified linear unit (ReLU) and a 2x2 max pooling operation with stride 2 for downsampling. At each downsampling step we double the number of feature channels. 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. The cropping is necessary due to the loss of border pixels in every convolution. At the final layer a 1x1 convolution is used to map each 64- component feature vector to the desired number of classes. In total the network has 23 convolutional layers. To allow a seamless tiling of the output segmentation map , it is important to select the input tile size such that all 2x2 max-pooling operations are applied to a layer with an even x- and y-size.

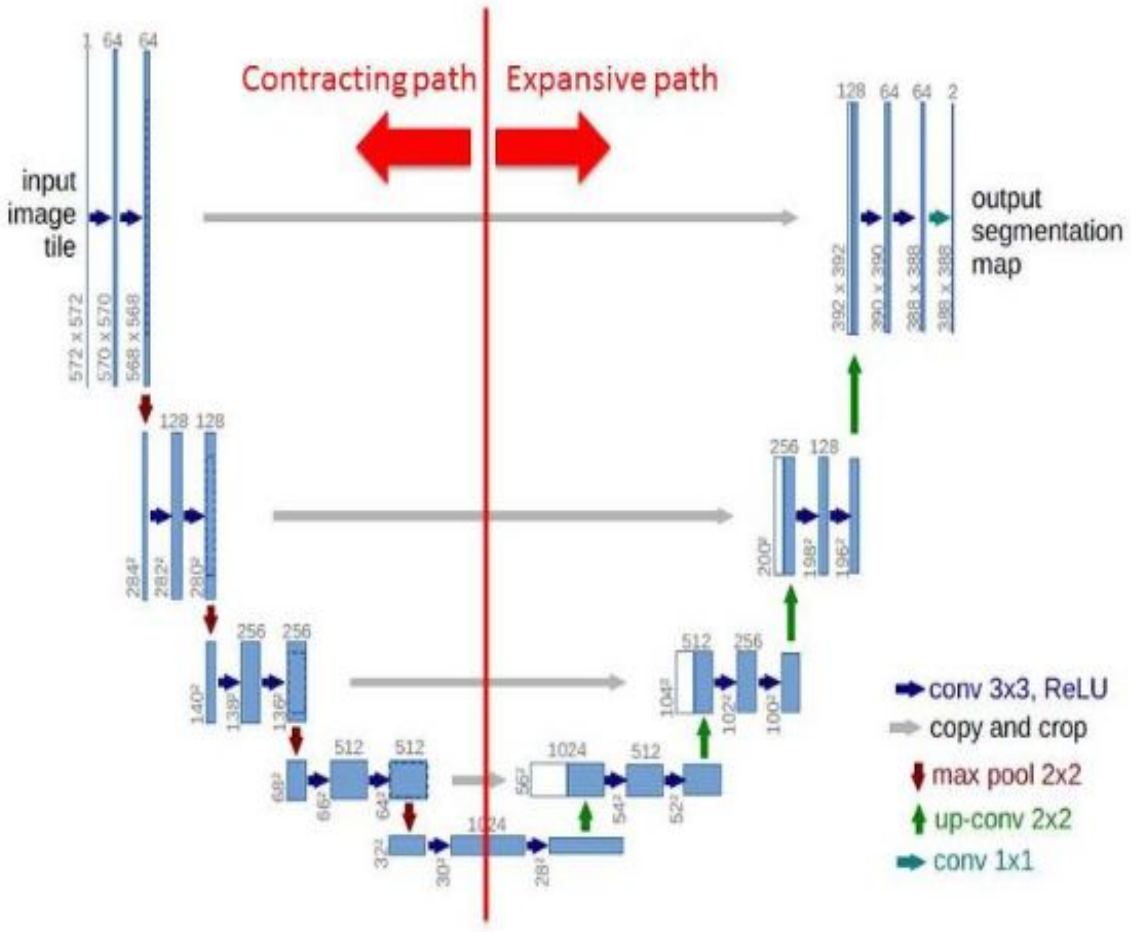


Figure 4.1 : Illustration of U-Net architecture (from U-Net paper)

4.1.1.1 Contracting/downsampling path

The contracting path is composed of 4 blocks. Each block is composed of

- 3x3 Convolution Layer + activation function (with batch normalization)
- 3x3 Convolution Layer + activation function (with batch normalization)
- 2x2 Max Pooling

Note that the number of feature maps doubles at each pooling, starting with 64 feature maps for the first block, 128 for the second, and so on. The purpose of this contracting path is to capture the context of the input image in order to be able to do segmentation. This coarse contextual information will then be transfered to the upsampling path by means of skip connections.

4.1.1.2 Bottleneck

This part of the network is between the contracting and expanding paths. The bottleneck is built from simply 2 convolutional layers (with batch normalization), with dropout.

4.1.1.3 Expanding/upsampling path

The expanding path is also composed of 4 blocks. Each of these blocks is composed of

- Deconvolution layer with stride 2
- Concatenation with the corresponding cropped feature map from the contracting path
- 3x3 Convolution layer + activation function (with batch normalization)
- 3x3 Convolution layer + activation function (with batch normalization)

The purpose of this expanding path is to enable precise localization combined with contextual information from the contracting path.

4.1.4 TRAINING

```
import tensorflow as tf
import numpy as np,sys,os
from sklearn.utils import shuffle
from scipy.ndimage import imread
import scipy.misc as smp
import matplotlib.pyplot as plt
import cv2
import keras

np.random.seed(678)
tf.set_random_seed(5678)



loss_func=[]

def tf_relu(x): return tf.nn.relu(x)
def d_tf_relu(s): return tf.cast(tf.greater(s,0),dtype=tf.float32)
def tf_softmax(x): return tf.nn.softmax(x)
def np_sigmoid(x): 1/(1 + np.exp(-1 *x))

# --- make class ---
class conlayer_left():

    def __init__(self,ker,in_c,out_c):
        self.w = tf.Variable(tf.random_normal([ker,ker,in_c,out_c],stddev=0.05))

    def feedforward(self,input,stride=1,dilate=1):
        self.input  = input
        self.layer  = tf.nn.conv2d(input,self.w,strides = [1,stride,stride,1],padding='SAME')
        self.layerA = tf_relu(self.layer)
```

```
        return self.layerA

class conlayer_right():

    def __init__(self,ker,in_c,out_c):
        self.w = tf.Variable(tf.random_normal([ker,ker,in_c,out_c],stddev=0.05))

    def feedforward(self,input,stride=1,dilate=1,output=1):
        self.input  = input

        current_shape_size = input.shape

        self.layer = tf.nn.conv2d_transpose(input,self.w,
        output_shape=[batch_size] +
[int(current_shape_size[1].value*2),int(current_shape_size[2].value*2),int(current_shape_
size[3].value/2)],strides=[1,2,2,1],padding='SAME')
        self.layerA = tf_relu(self.layer)
        return self.layerA

# --- get data ---
data_location = 'C:///Users///Jayanthi Mahendran///Desktop///Unet///samp'
train_data = []  # create an empty list

for dirName, subdirList, fileList in sorted(os.walk(data_location)):
    for filename in fileList:
        if '.png' in filename.lower():  # check whether the file's DICOM
            train_data.append(os.path.join(dirName,filename))

train_data_gt = []  # create an empty list
for x in train_data:
    train_data_gt.append(x[:-9]+".jpg")
```

```
print(str(len(train_data))+str(" ")+str(len(train_data_gt)))
#print(train_data[0])
#print(train_data_gt[0])
#
#print(train_data[1])
#print(train_data_gt[1])


train_images = np.zeros(shape=(18,256,256,1))
train_labels = np.zeros(shape=(18,256,256,1))

for file_index in range(len(train_data)):
    train_images[file_index,:,:]   =
np.expand_dims(smp.imresize(imread(train_data[file_index],mode='F',flatten=True),(256,
256)),axis=2)
    train_labels[file_index,:,:]   =
np.expand_dims(smp.imresize(imread(train_data_gt[file_index],mode='F',flatten=True),(2
56,256)),axis=2)

train_images = (train_images - train_images.min()) / (train_images.max() -
train_images.min())
train_labels = (train_labels - train_labels.min()) / (train_labels.max() - train_labels.min())

# --- hyper ---
num_epoch = 500
init_lr = 0.0001
batch_size = 2

# --- 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)

# ---- make graph ----
x = tf.placeholder(shape=[None,256,256,1],dtype=tf.float32)
y = tf.placeholder(shape=[None,256,256,1],dtype=tf.float32)

layer1_1 = l1_1.feedforward(x)
layer1_2 = l1_2.feedforward(layer1_1)
layer1_3 = l1_3.feedforward(layer1_2)

layer2_Input = tf.nn.max_pool(layer1_3,ksize=[1,2,2,1],strides=[1,2,2,1],padding='VALID')
layer2_1 = l2_1.feedforward(layer2_Input)
layer2_2 = l2_2.feedforward(layer2_1)
layer2_3 = l2_3.feedforward(layer2_2)

layer3_Input = tf.nn.max_pool(layer2_3,ksize=[1,2,2,1],strides=[1,2,2,1],padding='VALID')
layer3_1 = l3_1.feedforward(layer3_Input)
layer3_2 = l3_2.feedforward(layer3_1)
layer3_3 = l3_3.feedforward(layer3_2)

layer4_Input = tf.nn.max_pool(layer3_3,ksize=[1,2,2,1],strides=[1,2,2,1],padding='VALID')
layer4_1 = l4_1.feedforward(layer4_Input)
layer4_2 = l4_2.feedforward(layer4_1)
layer4_3 = l4_3.feedforward(layer4_2)

layer5_Input = tf.nn.max_pool(layer4_3,ksize=[1,2,2,1],strides=[1,2,2,1],padding='VALID')
layer5_1 = l5_1.feedforward(layer5_Input)
layer5_2 = l5_2.feedforward(layer5_1)
layer5_3 = l5_3.feedforward(layer5_2)
```

```
layer6_Input = tf.concat([layer5_3,layer5_Input],axis=3)
layer6_1 = l6_1.feedforward(layer6_Input)
layer6_2 = l6_2.feedforward(layer6_1)
layer6_3 = l6_3.feedforward(layer6_2)

layer7_Input = tf.concat([layer6_3,layer4_Input],axis=3)
layer7_1 = l7_1.feedforward(layer7_Input)
layer7_2 = l7_2.feedforward(layer7_1)
layer7_3 = l7_3.feedforward(layer7_2)

layer8_Input = tf.concat([layer7_3,layer3_Input],axis=3)
layer8_1 = l8_1.feedforward(layer8_Input)
layer8_2 = l8_2.feedforward(layer8_1)
layer8_3 = l8_3.feedforward(layer8_2)

layer9_Input = tf.concat([layer8_3,layer2_Input],axis=3)
layer9_1 = l9_1.feedforward(layer9_Input)
layer9_2 = l9_2.feedforward(layer9_1)
layer9_3 = l9_3.feedforward(layer9_2)

layer10 = l10_final.feedforward(layer9_3)

cost = tf.reduce_mean(tf.square(layer10-y))
auto_train = tf.train.AdamOptimizer(learning_rate=init_lr).minimize(cost)



accuracy_arr=[]

# --- start session ---
with tf.Session() as sess:
    saver = tf.train.Saver()
```

```
    sess.run(tf.global_variables_initializer())
    for iter in range(num_epoch):

        # train
        for current_batch_index in range(0,len(train_images),batch_size):
            current_batch =
train_images[current_batch_index:current_batch_index+batch_size,:,:,:]
            current_label =
train_labels[current_batch_index:current_batch_index+batch_size,:,:,:]
            sess_results =
sess.run([cost,auto_train],feed_dict={x:current_batch,y:current_label})
            print(' Iter: ', iter, " Cost:  %.32f"% sess_results[0],end='\r')
        print('\n----------------------')
        train_images,train_labels = shuffle(train_images,train_labels)
        loss_func.append(sess_results[0])
        if iter % 2 == 0:
            test_example =   train_images[:2,:,:,:]
            test_example_gt = train_labels[:2,:,:,:]
            sess_results = sess.run([layer10],feed_dict={x:test_example})

            sess_results = sess_results[0][0,:,:,:]
            test_example = test_example[0,:,:,:]
            test_example_gt = test_example_gt[0,:,:,:]




            originalMask=np.uint8(test_example*255)
            originalImage=np.uint8(test_example_gt*255)

originalOverlay=np.uint8(np.multiply(np.squeeze(test_example),np.squeeze(test_exampl
e_gt))*255)
```

```
generatedOverlay=np.uint8(np.multiply(np.squeeze(test_example),np.squeeze(sess_resu
lts))*255)
        generatedMask=np.uint8(np.squeeze(sess_results)*255);

        plt.figure();
        plt.imshow(np.squeeze(test_example_gt),cmap='gray')
        plt.axis('off')
        plt.title('Original Image')
        plt.savefig('train_change/'+str(iter)+"a_Original_Image.png")
        plt.close('all')

        plt.figure()
        plt.imshow(np.squeeze(test_example),cmap='gray')
        plt.axis('off')
        plt.title('Original Mask')
        plt.savefig('train_change/'+str(iter)+"b_Original_Mask.png")
        plt.close('all')

        plt.figure()
        plt.imshow(np.squeeze(sess_results),cmap='gray')
        plt.axis('off')
        plt.title('Generated Mask')
        plt.savefig('train_change/'+str(iter)+"c_Generated_Mask.png")
        plt.close('all')

        plt.figure()

plt.imshow(np.multiply(np.squeeze(test_example),np.squeeze(test_example_gt)),cmap='
gray')
        plt.axis('off')
        plt.title("Ground Truth Overlay")
```

```
        plt.savefig('train_change/'+str(iter)+"d_Original_Image_Overlay.png")
        plt.close('all')

        plt.figure()
        plt.axis('off')

plt.imshow(np.multiply(np.squeeze(test_example),np.squeeze(sess_results)),cmap='gray'
)
        plt.title("Generated Overlay")
        plt.savefig('train_change/'+str(iter)+"e_Generated_Image_Overlay.png")
        plt.close('all')

        target=np.multiply(np.squeeze(test_example),np.squeeze(test_example_gt))
        prediction=np.multiply(np.squeeze(test_example),np.squeeze(sess_results))
        intersection = np.logical_and(target, prediction)
        union = np.logical_or(target, prediction)
        iou_score = np.sum(intersection) / np.sum(union)
        accuracy_arr.append(iou_score)

mean=sum(accuracy_arr)
mean=mean/len(accuracy_arr)
print("MEAN ACCURACY:"+str(mean))

plt.figure()
plt.plot(np.arange(len(loss_func)),loss_func)
plt.xlabel('EPOCHS')
plt.ylabel('LOSS')
plt.title('AVERAGE MEAN SQUARE LOSS')
plt.show()
plt.savefig('cost_loss_plot.png')
plt.close('all')
```

-- end code --

4.1.5 ADVANTAGES

- This tool is really versatile and can be used for any reasonable image masking task.
- High accuracy given proper training, adequate dataset and training time.
- This architecture is input image size agnostic since it does not contain fully connected layers .
- This also leads to smaller model weight size .
- Can be easily scaled to have multiple classes.
- Relatively easy to understand why the architecture works, if you have basic understanding of how convolutions work.
- The U-Net combines the location information from the downsampling path with the contextual information in the upsampling path to finally obtain a general information combining localisation and context, which is necessary to predict a good segmentation map.
- No dense layer, so images of different sizes can be used as input (since the only parameters to learn on convolution layers are the kernel, and the size of the kernel is independent from input image' size).
- The use of massive data augmentation is important in domains like biomedical segmentation, since the number of annotated samples is usually limited.

4.1.6 DISADVANTAGES

- Because of many layers takes significant amount of time to train.
- Relatively high GPU memory footprint for larger images.
- The majority of Keras implementations are for outdated Keras versions.
- Is not standard to have pre-trained models widely available.

4.2 Convolutional Neural Network

Convolutional Neural Networks are very similar to ordinary Neural Networks. They are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other. And they still have a loss function (e.g. SVM/Softmax) on the last (fully-connected) layer and all the tips/tricks we developed for learning regular Neural Networks still apply.

Convolutional Neural Networks architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the amount of parameters in the network.

4.2.1 Layers in CNN

A simple Convolutional Neural Network is a sequence of layers, and every layer of a Convolutional Neural Networks transforms one volume of activations to another through a differentiable function. We use three main types of layers to build Convolutional Neural Networks architectures: **Convolutional Layer**, **Pooling Layer**, and **Fully-Connected Layer**. When these layers are stacked, a CNN architecture has been formed.

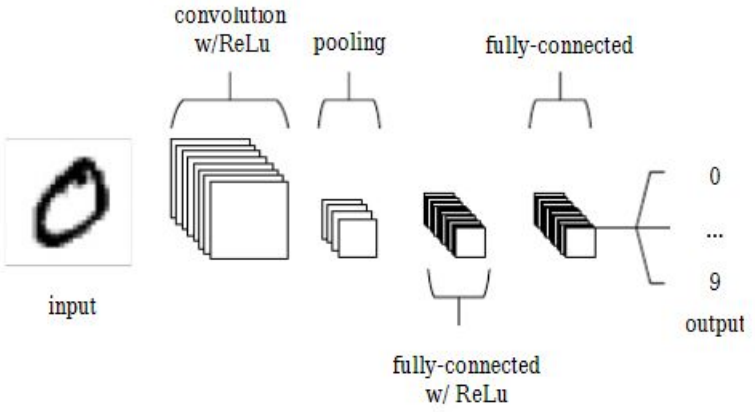


Fig. 4.2: An simple CNN architecture, comprised of just five layers

The basic functionality of the example CNN above can be broken down into four key areas.

1. As found in other forms of ANN, the input layer will hold the pixel values of the image.
2. The convolutional layer will determine the output of neurons of which are connected to local regions of the input through the calculation of the scalar product between their weights and the region connected to the input volume. The rectified linear unit (commonly shortened to ReLu) aims to apply an 'element wise' activation function such as sigmoid to the output of the activation produced by the previous layer.
3. The pooling layer will then simply perform down sampling along the spatial dimensionality of the given input, further reducing the number of parameters within that activation.
1. The fully-connected layers will then perform the same duties found in standard ANNs and attempt to produce class scores from the activations, to be used for classification. It is also suggested that ReLu may be used between these layers, as to improve performance.

4.2.1.1 Convolutional Layer

As the name implies, the convolutional layer plays a vital role in how CNNs operate. The layers parameters focus around the use of learnable kernels.
These kernels are usually small in spatial dimensionality, but spreads along the entirety of the depth of the input. When the data hits a convolutional layer, the layer convolves each filter across the spatial dimensionality of the input to produce a 2D activation map. As we glide through the input, the scalar product is calculated for each value in that kernel. From this the network will learn kernels that 'fire' when they see a specific feature at a given spatial position of the input. These are commonly known as activations

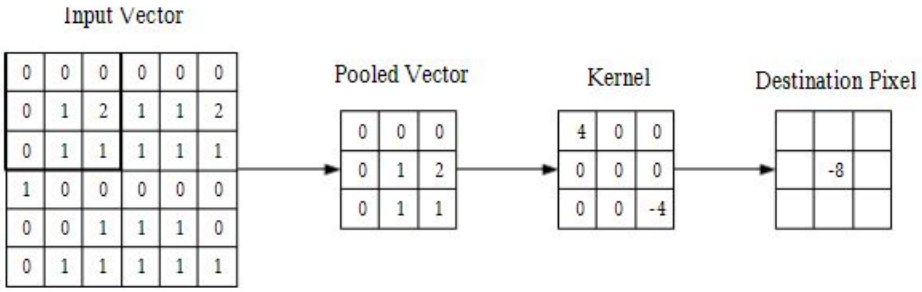


Fig. 4.3: A visual representation of a convolutional layer. The centre element of the kernel is placed over the input vector, of which is then calculated and replaced with a weighted sum of itself and any nearby pixels.

Every kernel will have a corresponding activation map, of which will be stacked along the depth dimension to form the full output volume from the convolutional layer.

Training ANNs on inputs such as images results in models of which are too big to train effectively .This comes down to the fully-connected manner of standard ANN neurons, so to mitigate against this every neuron in a convolutional layer is only connected to small region of the input volume. The dimensionality of this region is commonly referred to as the receptive field size of the neuron. The magnitude of the connectivity through the depth is nearly always equal to the depth of the input.

Convolutional layers are also able to significantly reduce the complexity of the model through the optimisation of its output. These are optimised through three hyper parameters, the **depth**, the **stride** and setting **zero-padding**.

The **depth** of the output volume produced by the convolutional layers can be manually set through the number of neurons within the layer to a the same region of the input. This can be seen with other forms of ANNs, where the all of the neurons in the hidden layer are directly connected to every single neuron beforehand. Reducing this hyper parameter can significantly minimise the total number of neurons of the network, but it can also significantly reduce the pattern recognition capabilities of the model.

We are also able to define the **stride** in which we set the depth around the spatial dimensionality of the input in order to place the receptive field. For example if we were to set a stride as 1, then we would have a heavily overlapped receptive field producing

extremely large activations. Alternatively, setting the stride to a greater number will reduce the amount of overlapping and produce an output of lower spatial dimensions.

Zero-padding is the simple process of padding the border of the input, and is an effective method to give further control as to the dimensionality of the output volumes.
It is important to understand that through using these techniques, it will alter the spatial dimensionality of the convolutional layers output. To calculate this, we can make use of the following formula :

$$\frac{(V - R) + 2Z}{S + 1}$$

Where V represents the input volume size (height× width × depth), R represents the receptive field size, Z is the amount of zero padding set and S referring to the stride. If the calculated result from this equation is not equal to a whole integer then the stride has been incorrectly set, as the neurons will be unable to fit neatly across the given input.

4.2.1.2 Pooling Layer

Pooling layers aim to gradually reduce the dimensionality of the representation, and thus further reduce the number of parameters and the computational complexity of the model. The pooling layer operates over each activation map in the input, and scales its dimensionality using the “MAX” function. In most CNNs, these come in the form of max-pooling layers with kernels of a dimensionality of 2 × 2 applied with a stride of 2 along the spatial dimensions of the input. This scales the activation map down to 25% of the original size - whilst maintaining the depth volume to its standard size.
It is also important to understand that beyond max-pooling, CNN architectures may contain general-pooling. General pooling layers are comprised of pooling neurons that are able to perform a multitude of common operations includingL1/L2-normalisation, and average pooling.

4.2.1.3 Fully-connected Layer

The fully-connected layer contains neurons of which are directly connected to the neurons in the two adjacent layers, without being connected to any layers within them. This is analogous to way that neurons are arranged in traditional forms of ANN.

4.2.2 Training

```
import pandas as pd
from sklearn import preprocessing
from keras.layers import Dense
from keras.models import Sequential
from sklearn.metrics import confusion_matrix
import numpy
from sklearn.model_selection import train_test_split

masses_data = pd.read_csv('info.csv')
masses_data = pd.read_csv('info.csv', na_values=['?'], names = ['BI-RADS', 'age',
'shape', 'margin', 'density', 'severity'])
all_features = masses_data[['age', 'shape','margin', 'density']].values;
all_classes = masses_data['severity'].values;
feature_names = ['age', 'shape', 'margin', 'density'];

scaler = preprocessing.StandardScaler()
all_features_scaled = scaler.fit_transform(all_features)

numpy.random.seed(1234)
(training_inputs,testing_inputs,training_classes,testing_classes) =
train_test_split(all_features_scaled, all_classes, train_size=0.75, random_state=1);

model = Sequential()
model.add(Dense(32, input_dim=4, kernel_initializer='normal', activation='relu'))
model.add(Dense(1, kernel_initializer='normal', activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='rmsprop', metrics=['accuracy'])
```

```
model.fit(training_inputs,training_classes,batch_size=10,nb_epoch=100);
y_pred=model.predict(testing_inputs);

y_pred = numpy.array(y_pred);
y_pred=(y_pred>0.5).astype(int);
print (y_pred);
"""
cm = confusion_matrix(testing_inputs, y_pred)
print(cm);"""
```

4.2.3 Regularization

Regularization is “any modification we make to the learning algorithm that is intended to reduce the generalization error, but not its training error”.
Regularization is a key component in preventing overfitting . Also, some techniques of regularization can be used to reduce model capacity while maintaining accuracy.
Some of the most common techniques of regularization used are:

- Dataset augmentation
- Early stopping
- Dropout layer
- Weight penalty L1 and L2

4.2.3.1 Dataset augmentation

An overfitting model (neural network or any other type of model) can perform better if learning algorithm processes more training data. While an existing dataset might be limited, for some machine learning problems there are relatively easy ways of creating synthetic data. For images some common techniques include translating the picture a few pixels, rotation, scaling. For classification problems it’s usually feasible to inject random negatives — e.g. unrelated pictures.
There is no general recipe regarding how the synthetic data should be generated and it varies a lot from problem to problem. The general principle is to expand the dataset by

applying operations which reflect real world variations as close as possible. Having better dataset in practice significantly helps quality of the models, independent of the architecture.

4.2.3.2 Early stopping

Early-stopping combats overfitting interrupting the training procedure once model's performance on a validation set gets worse. A validation set is a set of examples that we never use for gradient descent, but which is also not a part of the test set. The validation examples are considered to be representative of future test examples. Early stopping is effectively tuning the hyper-parameter number of epochs/steps.
Intuitively as the model sees more data and learns patterns and correlations, both training and test error go down. After enough passes over training data the model might start overfitting and learning noise in the given training set. In this case training error would continue going down while test error (how well we generalize) would get worse. Early stopping is all about finding this right moment with minimum test error.

4.2.3.3 Dropout layer

At each training iteration a dropout layer randomly removes some nodes in the network along with all of their incoming and outgoing connections. Dropout can be applied to hidden or input layer.
The nodes become more insensitive to the weights of the other nodes (co-adaptive), and therefore the model is more robust. If a hidden unit has to be working well with different combinations of other hidden units, it's more likely to do something individually useful.
Dropout can be viewed as a form of averaging multiple models ("ensemble"), technique which shows better performance in most machine learning tasks (e.g. ensemble training is the intuition behind random forests or gradient boosting decision trees). Training a neural network with dropout can be seen as training a collection of 2^n thinned networks with parameters sharing, where each thinned network gets trained very rarely, or not at all. Most of the thinned models, in fact, will never be used. Those which are used will likely get only one training example, which make it an extreme form of bagging. If you are

wondering how it can work, the trick is the sharing of the weights between all the models of this sample which means that each model is very strongly regularized by the others. With this method we don't need to train separate models which is in general pretty expensive, but we still get some of the benefits of ensemble methods.

4.2.3.4 Weight penalty L1 and L2

Weight penalty is standard way for regularization, widely used in training other model types. It relies strongly on the implicit assumption that a model with small weights is somehow simpler than a network with large weights. The penalties try to keep the weights small or non-existent (zero) unless there are big gradients to counteract it, which makes models also more interpretable.

L2 norm

- Penalizes the square value of the weight (which explains also the "2" from the name).
- Tends to drive all the weights to smaller values.

L1 norm

- Penalizes the absolute value of the weight (v- shape function).
- Tends to drive some weights to exactly zero (introducing sparsity in the model), while allowing some weights to be big.

CHAPTER 5

RESULT AND ANALYSIS

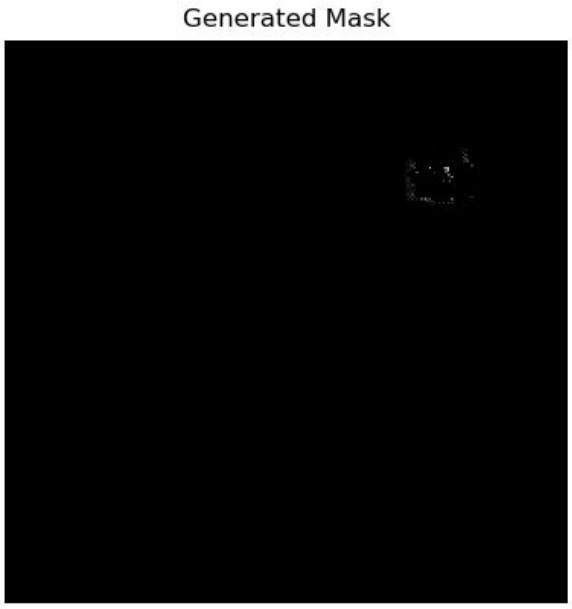
We have used training and testing data set (inbreast) of 106 images resizing them to 256 x 256 pixels.

The dataset has been trained for 500 epochs.

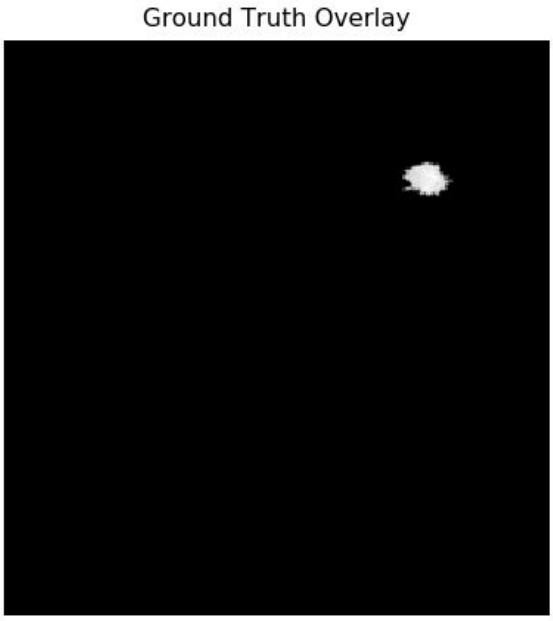
The following are the generated mask and the original image overlay obtained.

1.This has gone through 0 epochs.

a) Generated mask image:

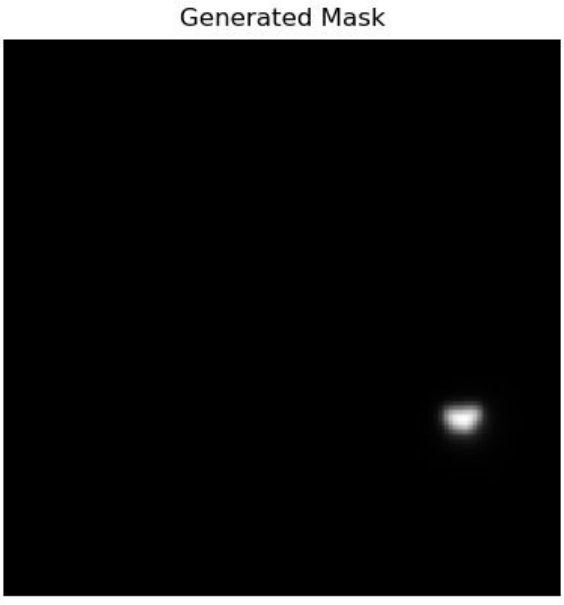


b) Original image overlay:

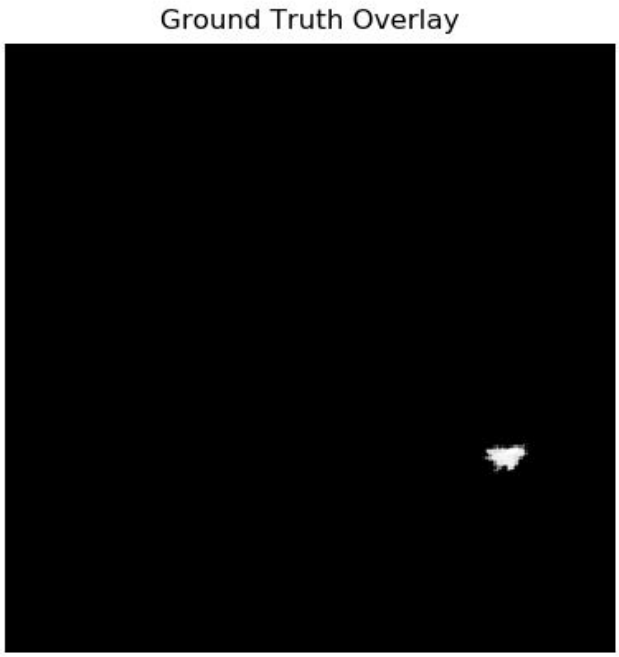


2.This has gone through 50 epochs.

a) Generated mask image:

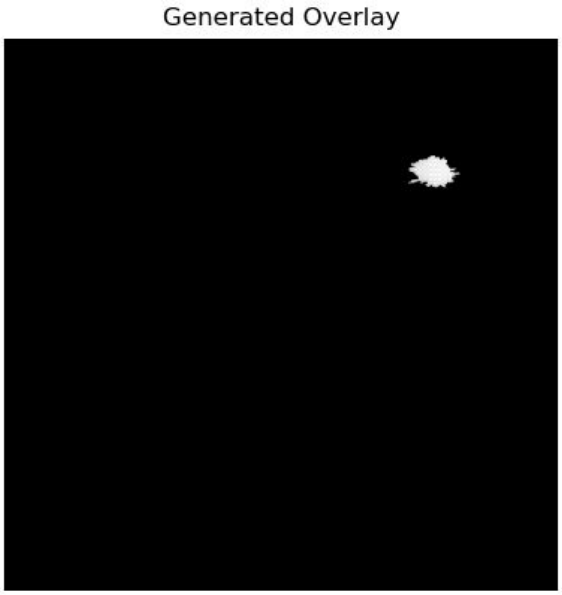


b) Original image overlay:

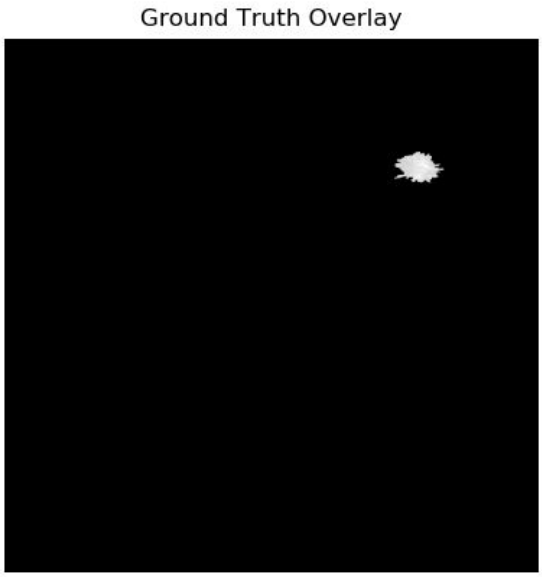


3.This has gone through 498 epochs.

a) Generated mask image:



b) Original image overlay:



REFERENCES:

[1] Olaf Ronneberger, Philipp Fischer, and Thomas Brox Computer Science Department and BIOSS Centre for Biological Signalling Studies, University of Freiburg, Germany, U-Net: Convolutional Networks for Biomedical Image Segmentation.

[2] John E Ball and Lori Mann Bruce. Digital mammographic computer aided diagnosis (cad) using adaptive level set segmentation. In 2007 29th Annual International Conference of the IEEE Engineering in Medicine and Biology Society, pages 4973–4978. IEEE, 2007.

[3] Wentao Zhu, Xiang Xiang, Trac D. Tran, Gregory D. Hager, Xiaohui Xie, ADVERSARIAL DEEP STRUCTURED NETS FOR MASS SEGMENTATION FROM MAMMOGRAMS

[4] Sachin Mehta , Ezgi Mercan , Jamen Bartlett , Donald Weaver , Joann G. Elmore , and Linda Shapiro, Y-Net: Joint Segmentation and Classification for Diagnosis of Breast Biopsy Images

[5] Peter Boyle, Bernard Levin, et al. World cancer report 2008. IARC Press, International Agency for Research on Cancer, 2008

[6] Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, Gradient-Based Learning Applied to Document Recognition

[7] Carol DeSantis, Jiemin Ma, Leah Bryan, and Ahmedin Jemal. Breast cancer statistics, 2013. CA: a cancer journal for clinicians, 64(1):52–62, 2014.

[8] Thijs Kooi , Geert Litjens , Bram van Ginnekena , Albert Gubern-Méridaa , Clara I. Sánchez a , Ritse Manna , Ard den Heetenb , Nico Karssemeijer, Large scale deep learning for computer aided detection of mammographic lesions.

[9] Fei Gao, Teresa Wu, Jing Li, Bing Zheng, Linxiang Ruan, Desheng Shang and Bhavika Patel, SD-CNN: a Shallow-Deep CNN for Improved Breast Cancer Diagnosis.

[10] Jonathan Long, Evan Shelhamer, Trevor Darrell, UC Berkeley, Fully Convolution Neural Networks for Semantic Segmentation.

[11] Chen Wang, Yang Xi, Convolutional Neural Network for Image Classification.