BINGHAMTON UNIVERSITY | The State University of New York

Watson Graduate School of Engineering & Applied Science
Department of Electrical & Computer Engineering

Neural Network & Deep Learning (EECE680C) Project

"Retinal Optical Coherence Tomography (OCT)"

Group Members	Alem Fitwi Charlie Miller, Xiaojing Xia
submitted to	Prof. *Xiaohua Li*
Due Date	May 17, 2018

We did it collaboratively! We had a lot of meetings ftom start to end!

Outline/Content:

I. Introduction

Convolutional Neural Networks:

- CNN is made up of neurons that have learnable weights and biases.

- Best for gridded source data like image that allows to encode certain properties into the architecture.

- Neurons arranged in 3 dimensions: **width, height, depth.**

- The main layers/processes are:
    - **Convolutional Layer + Activation**
    - **Pooling Layer**
    - **Flattening**
    - **Fully-Connected Layer**

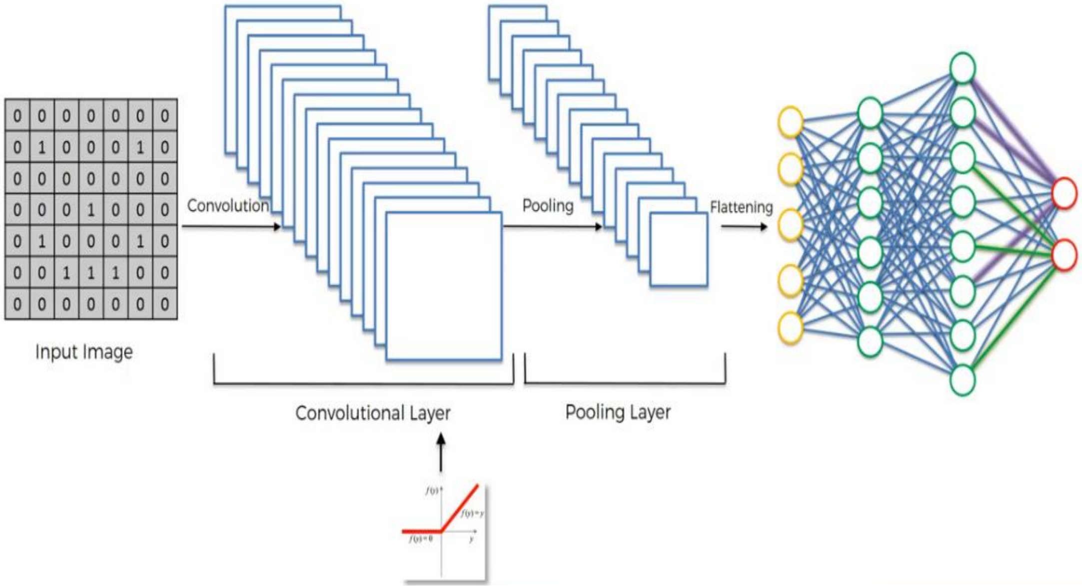


Fig-1: Basic Architecture of CNN

… Introduction

- ***INPUT***: holds the raw pixel values of the image.
- ***CONV layer:*** computes the output of neurons that are connected to local regions in the input.
- ***RELU layer:*** activation function
- ***POOLING layer:*** performs a downsampling operation along the spatial dimensions(width, height)
- ***Fully Connected layer***: computes the class scores

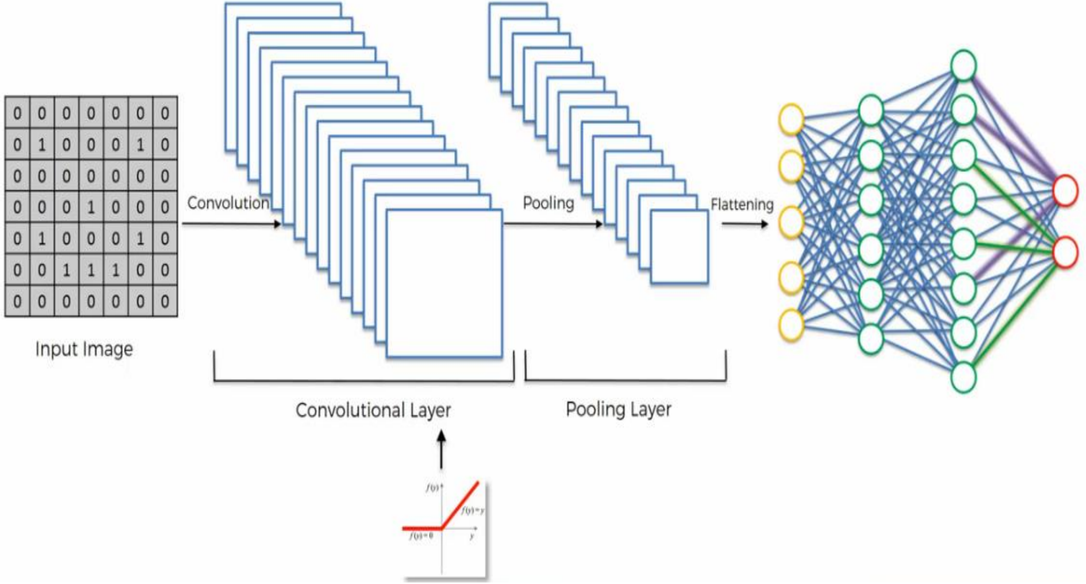


Fig-1: Basic Architecture of CNN

II. Dataset - Motivation

Fig-2: Analyzing Retina Image

Background:

- Diabetic macular edema (DME) and Choroidal Neovascularization (CNV) associated with age-related macular degeneration (AMD) are the leading causes of vision loss in the industrialized world.
- Without prompt medical innervation, many will experience universal vision problems.
- Many people with these diseases are under the condition to access to specialist for training.
- At the same time, many in undeveloped urban areas cannot find a quick treatment due to high patient volume. A ***Doctor is not everywhere.***
- **Low cost**
- **Convenient**

- It effectively classified images for macular degeneration and diabetic retinopathy
- It also accurately distinguished bacterial and viral pneumonia on chest X-rays
- This has potential for generalized high-impact application in biomedical imaging

… Dataset

- Retinal optical coherence tomography (OCT) - Scans of Retinas
- 30 million performed each year
- 4 classes - NORMAL, CNV, DME, DRUSEN (AMD)
- 84,452 images labeled by experts
- Distinguishing features indicated by arrows

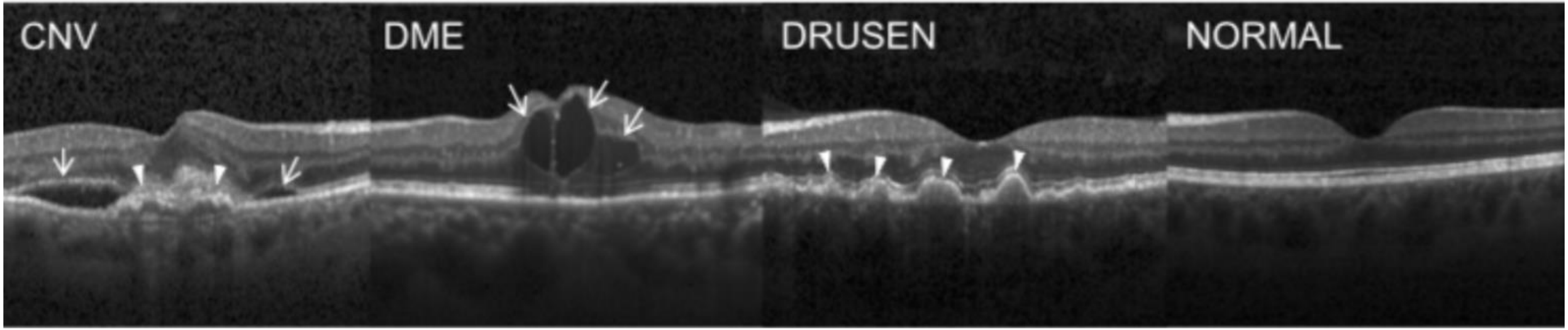
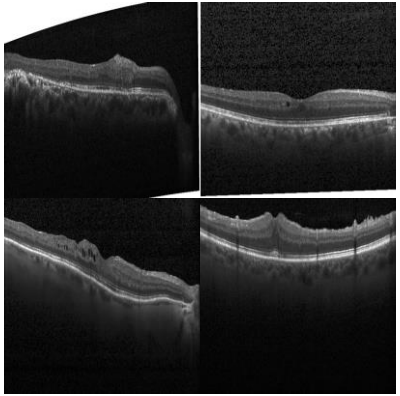
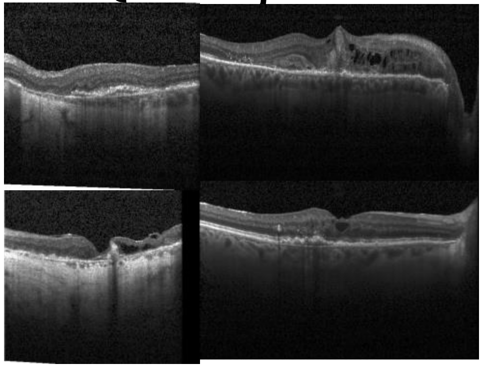


Fig-3: The four classes of Retina OCT Images

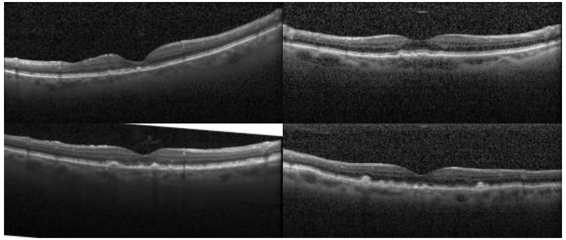
... Dataset → Examples of Each Class (Irregular Shapes)



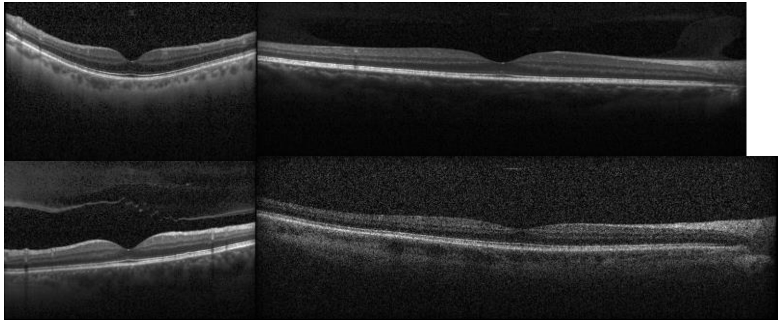
4 DME Images: Holes in the middle



4 CNV Images: Build-up in the middle and holes on the sides



4 DRUSSEN Images : small bumps in middle



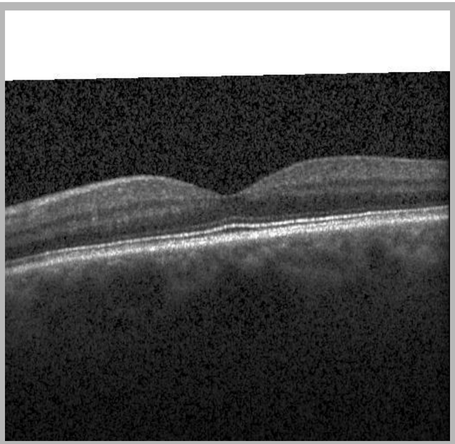
4 NORMAL Images: No irregular features

Fig-4: Examples of each class image

... Dataset

Example Image

- Histogram of image shown bottom right
  - Shows count of pixels per intensity (0-255, white being the brightest)
- Most entropy found in the 25 - 100 intensity
- CNN is most sensitive to these values



Example Normal Retina image

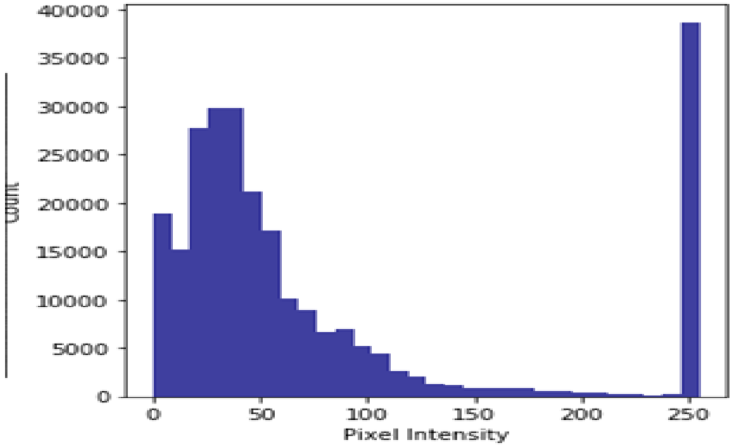


Fig-5: Image Histogram

... Dataset

Dataset Structure

- Each category has sub-groups
- Images are grouped together by a 6-digit number
  - Images in same group look similar
- Our first dataset did not take this into consideration
- In final dataset, we took 2057 images from each class
  - We took 1 to 4 from each subgroup
  - Images more varied this way
  - Images of same group in blue box
  - 2 distinct group images indicated by red box

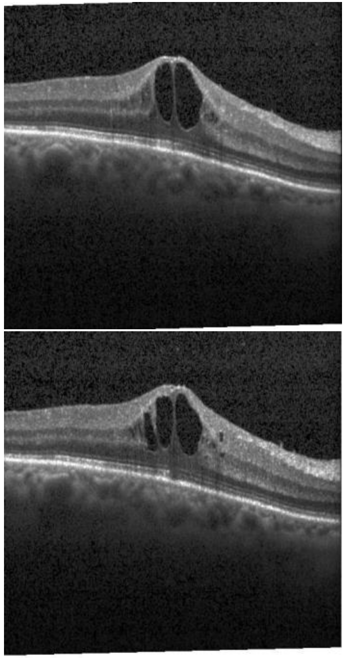
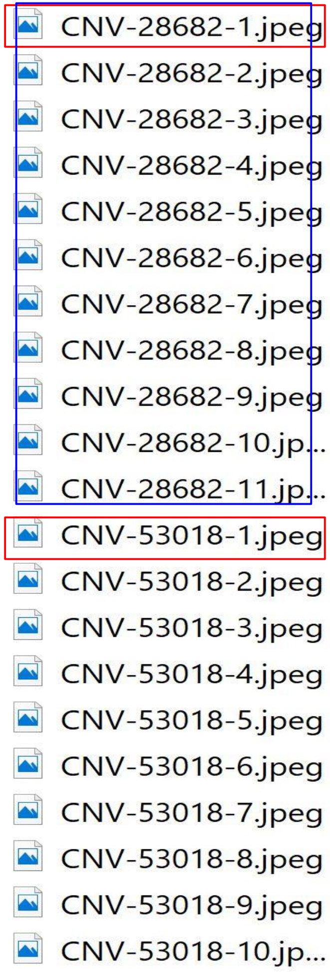


Fig-6: Similar images - same patient



III. Architecture

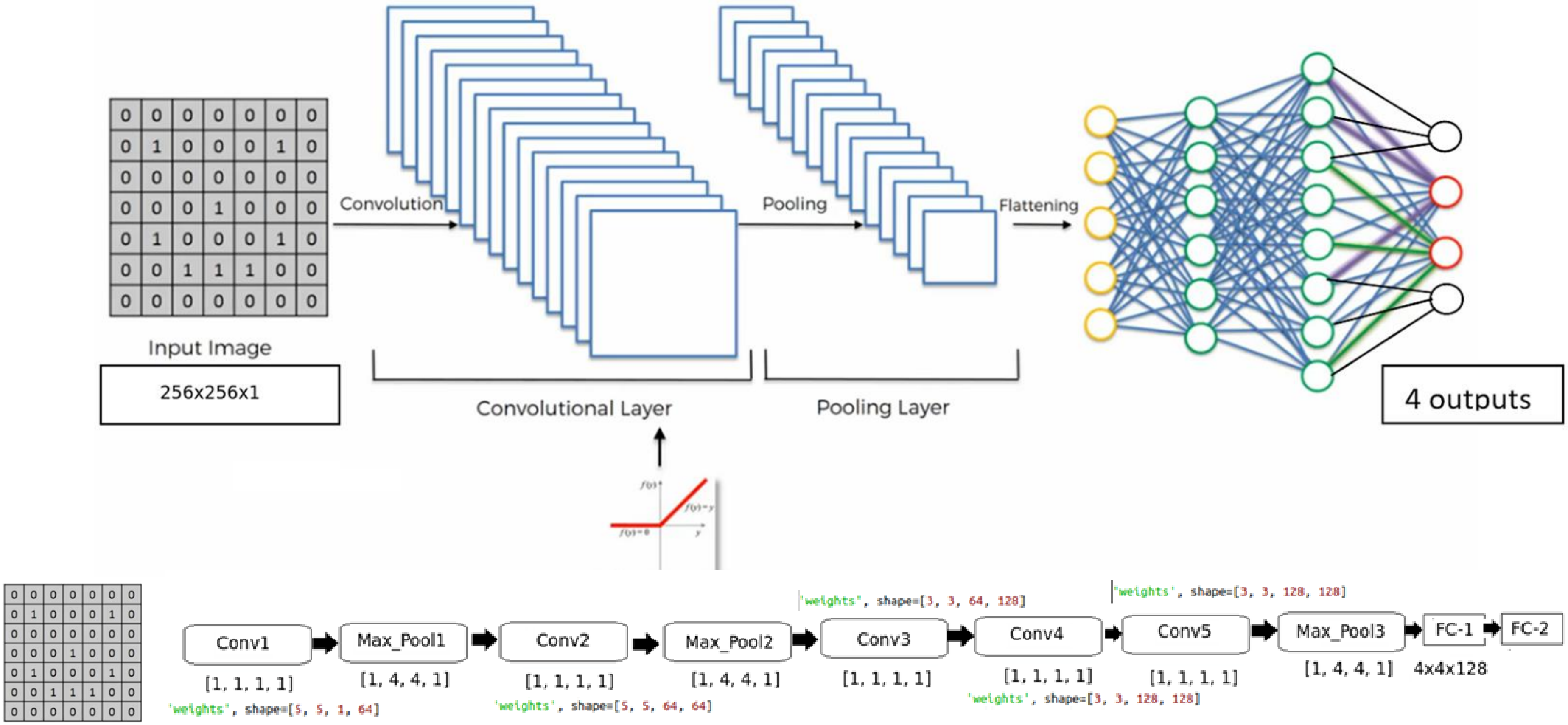


Fig-6: Our CNN Architecture

... Architecture

- ★ Image Size=256x256 (grayscale)
- ★ Test Images: 968
- ★ Training Data: 83,484
- ★ Our Architecture Comprises
  - > 5 convolutional layers of Kernel=[1, 1, 1, 1]
  - > 3 Max_pooling Layers of Kernel=[1, 4, 4, 1]
  - > 2 Fully Connected Layers of 384 and 192 neurons
- ★ All of the above hyperparameters were determined after many attempts or trainings.
- ★ These hyperparameters and corresponding weight shapes give the best results of all attempts we made.

... Architecture

- ★ Fig-7 is the architecture that was used by the people who developed these dataset. They used a transfer Network.
- ★ They didn't develop their own DL Architecture. Rather, they used a transfer network to make their input dataset suitable to google net.

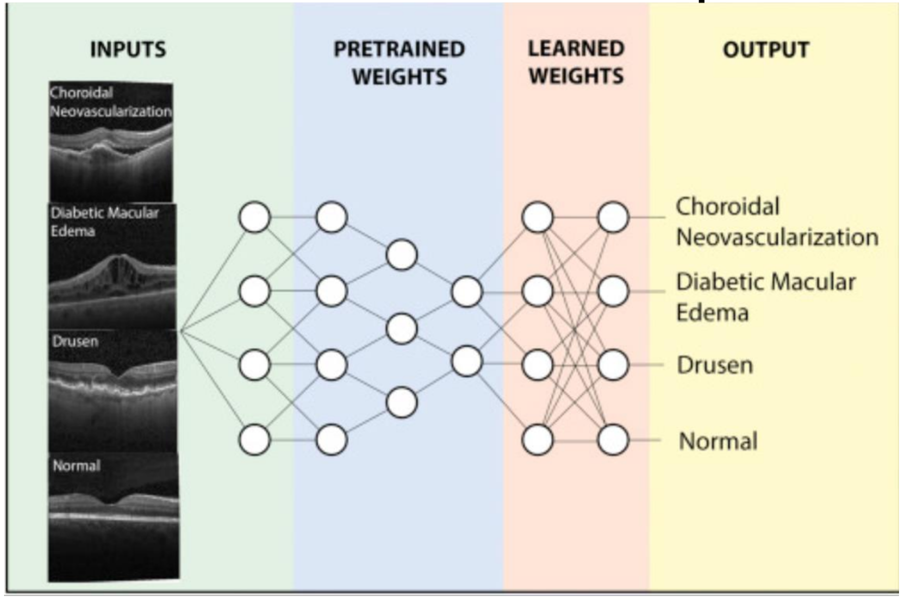
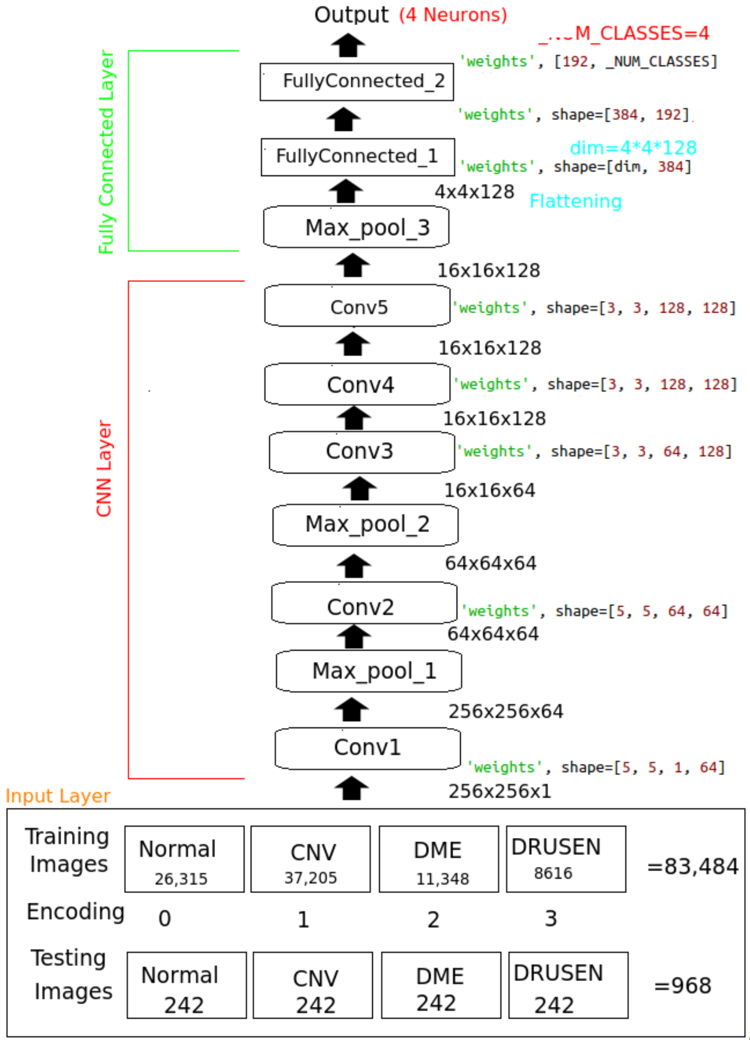


Fig-7: This is their model

IV. Shape of Data

After a number of trainings using various settings, we have settled on the following parameters that gave us very good result:

- ★ INPUT Shape = [256x256x1] → Image size is reduced to 256x256 to reduce the computational time.
- ★ CONV layers: each computes a dot product between their weights and a small region they are connected to in the input volume. Conv1 & 2 Use 64 filters, Conv3, 4, & 5 use 128 filters for each.
- ★ Max Pooling layer: performs a downsampling operation(4x4) , kernel=[1,4,4,1]
- ★ Filter sizes of conv1 → [5 5 1 64], Conv2 → [5 5 64 64], and filter sizes of Conv3 , Conv4, & Conv5→ [3 3 64 128] were adopted after many attempts.
- ★ They happened to give us the best results. Previously we tried [1 2 2 1] size filters, but the training was too slow.



V. Implementation/Coding

★ The most important parts of our python program are hereunder briefly presented.

This method reads and loads the training and testing images from the folders they are placed and encodes them as per the class they belong to. There four classes of images namely:

- ➔ ***Normal encoded as 0***
- ➔ ***CNV encoded as 1***
- ➔ ***DME encoded as 2***
- ➔ ***DRUSEN encoded as 3***

```python
#Load Training and Testing Datasets from the newly created folders
def get_data(folder):
    """
    Load the data and labels from the given folder.
    """
    X = []
    y = []
    for folderName in os.listdir(folder):
        if not folderName.startswith('.'):
            if folderName in ['NORMAL']:
                label = 0
            elif folderName in ['CNV']:
                label = 1
            elif folderName in ['DME']:
                label = 2
            elif folderName in ['DRUSEN']:
                label = 3
            for image_filename in tqdm(os.listdir(folder + folderName)):
                img_file = cv2.imread(folder + folderName + '/' +
                                      image_filename)
                if img_file is not None:
                    img_file = skimage.transform.resize(img_file,
                                                        (imageSize, imageSize, 1))
                    img_arr = np.asarray(img_file)
                    X.append(img_arr)
                    y.append(label)
    X = np.asarray(X)
    y = np.asarray(y)
    return X,y
```

...Implementation/Coding

The first convolutional

and Max_pooling ayers

```
    #---------------------------------------------------------------------------
    #Convolution Layer_1
    with tf.variable_scope('conv1') as scope:
        #Create kernel - 5x5 size, 64 outpupts
        kernel = variable_with_weight_decay('weights', shape=[5, 5, 1, 64],
                                            stddev=5e-2, wd=0.0)
        #Do the convolution
        conv = tf.nn.conv2d(x_image, kernel, [1, 1, 1, 1], padding='SAME')
        biases = variable_on_cpu('biases', [64], tf.constant_initializer(0.0))
        pre_activation = tf.nn.bias_add(conv, biases)
        #Compute output of layer
        conv1 = tf.nn.relu(pre_activation, name=scope.name)

    #Log results
    tf.summary.histogram('Convolution_layers/conv1', conv1)
    tf.summary.scalar('Convolution_layers/conv1', tf.nn.zero_fraction(conv1))

    #normalize the data
    norm1 = tf.nn.lrn(conv1, 4, bias=1.0, alpha=0.001 / 9.0, beta=0.75,
                                                      name='norm1')
    #---------------------------------------------------------------------------
    #Max_pool_1
    pool1 = tf.nn.max_pool(norm1, ksize=[1, 3, 3, 1], strides=[1, 4, 4, 1],
                                        padding='SAME', name='pool1')
```

...Implementation/Coding

Conv2, Max_pool_2,

and Conv3

```
#------------------------------------------------------------------------
#Convolution Layer_2 - 5x5, output 64 channels
with tf.variable_scope('conv2') as scope:
    kernel = variable_with_weight_decay('weights', shape=[5, 5, 64, 64],
                                        stddev=5e-2, wd=0.0)
    conv = tf.nn.conv2d(pool1, kernel, [1, 1, 1, 1], padding='SAME')
    biases = variable_on_cpu('biases', [64], tf.constant_initializer(0.1))
    pre_activation = tf.nn.bias_add(conv, biases)
    conv2 = tf.nn.relu(pre_activation, name=scope.name)
tf.summary.histogram('Convolution_layers/conv2', conv2)
tf.summary.scalar('Convolution_layers/conv2', tf.nn.zero_fraction(conv2))

norm2 = tf.nn.lrn(conv2, 4, bias=1.0, alpha=0.001 / 9.0, beta=0.75,
                  name='norm2')
#------------------------------------------------------------------------
#Max_pool_2
pool2 = tf.nn.max_pool(norm2, ksize=[1, 3, 3, 1], strides=[1, 4, 4, 1],
                       padding='SAME', name='pool2')
#------------------------------------------------------------------------
#Convolution Layer_3, 3x3,128 output channels
#We will not max-pool the next few layers since the images are
#sufficiently small to process
with tf.variable_scope('conv3') as scope:
    kernel = variable_with_weight_decay('weights', shape=[3, 3, 64, 128],
                                        stddev=5e-2, wd=0.0)
    conv = tf.nn.conv2d(pool2, kernel, [1, 1, 1, 1], padding='SAME')
    biases = variable_on_cpu('biases', [128], tf.constant_initializer(0.0))
    pre_activation = tf.nn.bias_add(conv, biases)
    conv3 = tf.nn.relu(pre_activation, name=scope.name)
tf.summary.histogram('Convolution_layers/conv3', conv3)
tf.summary.scalar('Convolution_layers/conv3', tf.nn.zero_fraction(conv3))
```

...Implementation/Coding

Conv4, Conv5,
and

Max_pool_3

```
#---------------------------------------------------------------------------
#Convolution Layer_4, 3x3, 128 output channels, no max-pool
with tf.variable_scope('conv4') as scope:
    kernel = variable_with_weight_decay('weights', shape=[3, 3, 128, 128],
                                        stddev=5e-2, wd=0.0)
    conv = tf.nn.conv2d(conv3, kernel, [1, 1, 1, 1], padding='SAME')
    biases = variable_on_cpu('biases', [128], tf.constant_initializer(0.0))
    pre_activation = tf.nn.bias_add(conv, biases)
    conv4 = tf.nn.relu(pre_activation, name=scope.name)
tf.summary.histogram('Convolution_layers/conv4', conv4)
tf.summary.scalar('Convolution_layers/conv4', tf.nn.zero_fraction(conv4))
#---------------------------------------------------------------------------
#Convolution Layer_5,3x3, 128 output channels, WITH max-pool
with tf.variable_scope('conv5') as scope:
    kernel = variable_with_weight_decay('weights', shape=[3, 3, 128, 128],
                                        stddev=5e-2, wd=0.0)
    conv = tf.nn.conv2d(conv4, kernel, [1, 1, 1, 1], padding='SAME')
    biases = variable_on_cpu('biases', [128], tf.constant_initializer(0.0))
    pre_activation = tf.nn.bias_add(conv, biases)
    conv5 = tf.nn.relu(pre_activation, name=scope.name)
tf.summary.histogram('Convolution_layers/conv5', conv5)
tf.summary.scalar('Convolution_layers/conv5', tf.nn.zero_fraction(conv5))

norm3 = tf.nn.lrn(conv5, 4, bias=1.0, alpha=0.001 / 9.0, beta=0.75,
                  name='norm3')
#---------------------------------------------------------------------------
#Max_pool_3
pool3 = tf.nn.max_pool(norm3, ksize=[1, 3, 3, 1], strides=[1, 4, 4, 1],
                       padding='SAME', name='pool3')
```

...Implementation/Coding

Fully connected layers 1 and 2

```
#-----------------------------------------------------------------------
#Hold 2 fully connected layers - more depth
#Fully Connected Layer_1 - 384 outputs
with tf.variable_scope('fully_connected1') as scope:
    #reshape the convolution layer outputs to a flat vector
    reshape = tf.reshape(pool3, [-1, _RESHAPE_SIZE])
    dim = reshape.get_shape()[1].value
    #Get weights and biases
    weights = variable_with_weight_decay('weights', shape=[dim, 384],
                                         stddev=0.04, wd=0.004)
    biases = variable_on_cpu('biases', [384], tf.constant_initializer(0.1))
    #Apply non-linear function (Rectified Linear Unit)
    local3 = tf.nn.relu(tf.matmul(reshape, weights) + biases,
                        name=scope.name)
#Log results
tf.summary.histogram('Fully connected layers/fc1', local3)
tf.summary.scalar('Fully connected layers/fc1',
                  tf.nn.zero_fraction(local3))
#-----------------------------------------------------------------------
#Fully Connected Layer_2 - 192 outputs
with tf.variable_scope('fully_connected2') as scope:
    weights = variable_with_weight_decay('weights', shape=[384, 192],
                                         stddev=0.04, wd=0.004)
    biases = variable_on_cpu('biases', [192], tf.constant_initializer(0.1))
    local4 = tf.nn.relu(tf.matmul(local3, weights) + biases,
                        name=scope.name)
tf.summary.histogram('Fully connected layers/fc2', local4)
tf.summary.scalar('Fully connected layers/fc2',
                  tf.nn.zero_fraction(local4))
```

...Implementation/Coding

Output Layer

```
#-----------------------------------------------------------------------
#Output layer - condensce FC layer to 4 output neurons
#One for each class
with tf.variable_scope('output') as scope:
    weights = variable_with_weight_decay('weights', [192, _NUM_CLASSES],
                                         stddev=1 / 192.0, wd=0.0)
    biases = variable_on_cpu('biases', [_NUM_CLASSES],
                             tf.constant_initializer(0.0))

    #Put output layer through a softmax layer
    softmax_linear = tf.add(tf.matmul(local4, weights), biases,
                            name=scope.name)
#-----------------------------------------------------------------------
tf.summary.histogram('Fully connected layers/output', softmax_linear)

global_step = tf.Variable(initial_value=0, name='global_step',
                          trainable=False)
#State which class the neural network thinks the
#image looks like the most
y_pred_cls = tf.argmax(softmax_linear, axis=1)

return x, y, softmax_linear, global_step, y_pred_cls
```

VI. Result and Discussion

- ★ Before finding the best architecture, we have made a number of different settings and trainings.
- ★ We start with a simple architecture comprising two conv layers and one Max pooling layer → but the performance was very poor.
- ★ Then, we trained an architecture comprising 4 convolutional layers and two Max pooling layers. But we were not pleased by the performance yet.
- ★ Eventually, we adopted a deeper architecture or model comprising 5 convolutional layers and three Max pooling layers that gives a very good performance.

... Result and Discussion

- ★ Due to the bigger image size (typically 512x496 and 168x512 and too many images (more than 85 thousands of images) of the dataset, performing the training using our laptops or machines was ***next to impossible***.
- ★ We had to do a lot of analysis and input preprocessing to come up with a reduced representative dataset that could be effectively trained using our model.
- ★ Then, we settled at a reduced dataset of 8,228 for training and 300 for testing.
- ★ We did our training and testing for a reduced dataset on ***Google Colaboratory***.

… Result and Discussion

- ★ The table on the right side portrays the results we collected for various settings on ***Google Colaboratory*** that led us to the final and stable model that can do classification with an accuracy of 90% to 96%

Layers & Dataset	**Number or size**	**Training Accuracy**	**Test Accuracy**	**Remark**
Training Images	400	**57%**	**23%**	Image size=512x512x1
Testing Images	120			Image size=512x512x1
Convolutional	2			Kernel=[1, 2, 2, 1]
Max Pooling	1			
Training Images	1600	**75%**	**61%**	Image size=512x512x1
Testing Images	120			Image size=512x512x1
Convolutional	4			Kernel=[1, 2, 2, 1]
Max Pooling	2			Kernel=[1, 4, 4, 1]
Training Images	8,228	**96.2%** (Average)	**90.33%** (Average)	Image size=256x256x1
Testing Images	300			Image size=256x256x1
Convolutional	5			Kernel=[1, 1, 1, 1]
Max Pooling	3			Kernel=[1, 4, 4, 1]

... Result and Discussion

★ Here is a snapshot of our result as run on ***Google Colaboratory***

```
Iteration 1209Global Step:   1210, accuracy:  96.0%, loss = 0.12 (2.0 examples/sec, 25.45 sec/batch)
Iteration 1219Global Step:   1220, accuracy:  94.0%, loss = 0.17 (2.0 examples/sec, 25.41 sec/batch)
Iteration 1229Global Step:   1230, accuracy:  98.0%, loss = 0.12 (2.0 examples/sec, 25.03 sec/batch)
Iteration 1239Global Step:   1240, accuracy:  98.0%, loss = 0.06 (2.0 examples/sec, 24.89 sec/batch)
Iteration 1249Global Step:   1250, accuracy:  92.0%, loss = 0.20 (2.0 examples/sec, 24.96 sec/batch)
Iteration 1259Global Step:   1260, accuracy: 100.0%, loss = 0.08 (2.0 examples/sec, 24.98 sec/batch)
Iteration 1269Global Step:   1270, accuracy:  96.0%, loss = 0.21 (2.0 examples/sec, 25.24 sec/batch)
Iteration 1279Global Step:   1280, accuracy: 100.0%, loss = 0.05 (2.0 examples/sec, 25.30 sec/batch)
Iteration 1289Global Step:   1290, accuracy:  96.0%, loss = 0.14 (2.0 examples/sec, 24.88 sec/batch)
Iteration 1299Global Step:   1300, accuracy:  98.0%, loss = 0.13 (2.0 examples/sec, 25.32 sec/batch)
Accuracy on Test-Set: 96.33% (289 / 300)
```

Test result at 1300 iterations

... Result and Discussion

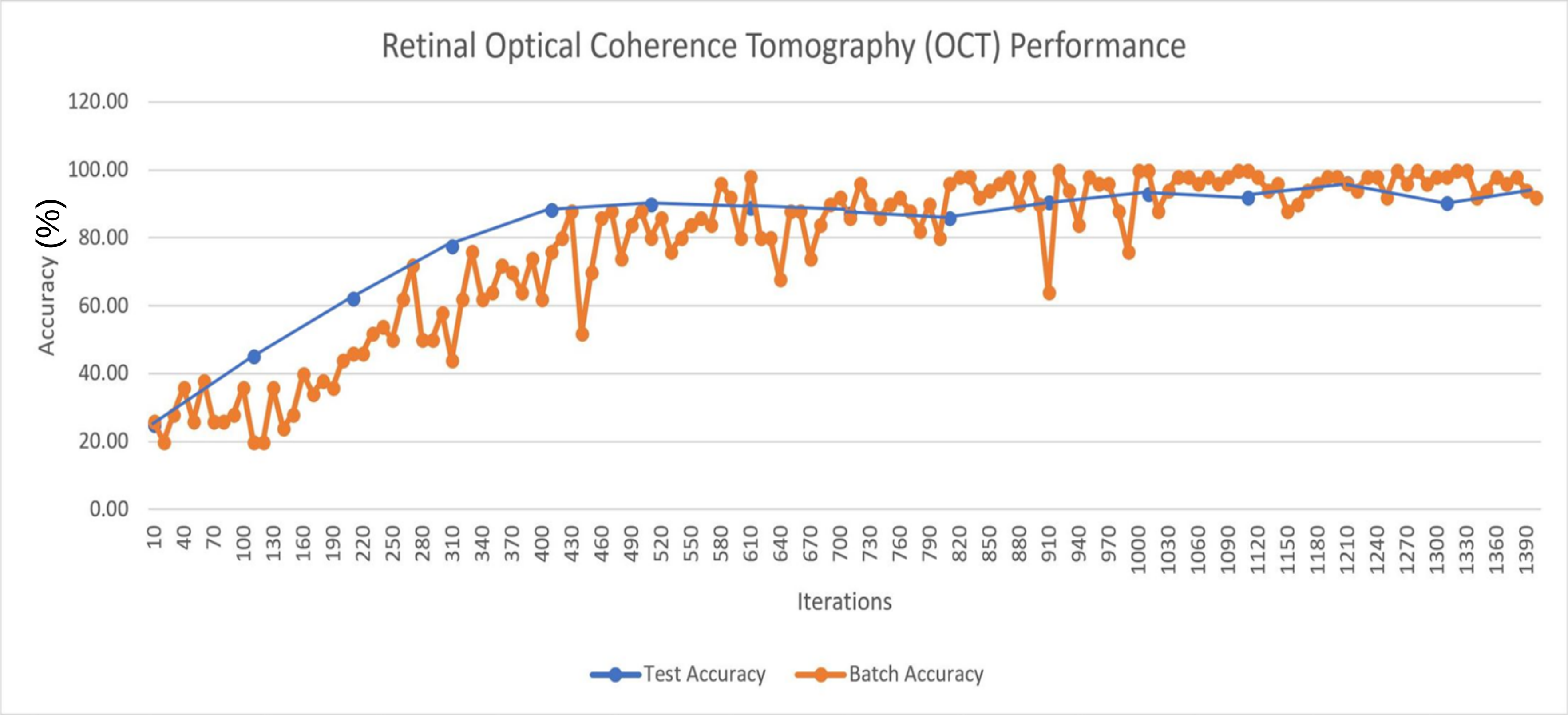
★ Here is a snapshot of our result as run on ***Google Colaboratory***

```
Iteration 1309Global Step:   1310, accuracy:  98.0%, loss = 0.07 (2.0 examples/sec, 25.47 sec/batch)
Iteration 1319Global Step:   1320, accuracy: 100.0%, loss = 0.06 (2.0 examples/sec, 25.27 sec/batch)
Iteration 1329Global Step:   1330, accuracy: 100.0%, loss = 0.08 (2.0 examples/sec, 25.10 sec/batch)
Iteration 1339Global Step:   1340, accuracy:  92.0%, loss = 0.19 (2.0 examples/sec, 25.36 sec/batch)
Iteration 1349Global Step:   1350, accuracy:  94.0%, loss = 0.16 (2.0 examples/sec, 25.22 sec/batch)
Iteration 1359Global Step:   1360, accuracy:  98.0%, loss = 0.10 (2.0 examples/sec, 25.11 sec/batch)
Iteration 1369Global Step:   1370, accuracy:  96.0%, loss = 0.11 (2.0 examples/sec, 24.61 sec/batch)
Iteration 1379Global Step:   1380, accuracy:  98.0%, loss = 0.07 (2.0 examples/sec, 25.10 sec/batch)
Iteration 1389Global Step:   1390, accuracy:  94.0%, loss = 0.17 (2.0 examples/sec, 25.23 sec/batch)
Iteration 1399Global Step:   1400, accuracy:  92.0%, loss = 0.23 (2.0 examples/sec, 24.92 sec/batch)
Accuracy on Test-Set: 90.33% (271 / 300)
```

Test result at 1400 iterations

… Result and Discussion

Graphical depiction of the training and testing performance of our CNN model



VI. CONCLUSION

- ★ Due the facts that the dataset comprises too many Retina OCT images (***84,452***) of irregularly larger size (***some 496x512 and others 768x512***), doing the training and determining the optimal hyperparameters were very challenging for us for we limited computing resources.
- ★ But after doing a great deal of analyses on the dataset and the apropos hyperparameters, we have managed to come up with a CNN model that can effectively classify the Retina OCT images with an accuracy of 90% to 96%.
- ★ We did all our trainings and testings on ***Google Colaboratory*** *with reduced image size (256x256) and reduced but representative dataset (8,228)*

VII. References

[1] https://adeshpande3.github.io/adeshpande3.github.io/The-9-Deep-Learning-Papers-You-Need-To-Know-About.html

[2] https://www.superdatascience.com/deep-learning/

[3] http://http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf

[4] https://arxiv.org/pdf/1609.04112.pdf

[5] http://ais.uni-bonn.de/papers/icann2010_maxpool.pdf

[6] https://www.udemy.com/deeplearning/learn/v4/questions/2276518

[7] https://www.kaggle.com/kmader/detect-retina-damage-from-oct-images-hr

[8] https://www.kaggle.com/paultimothymooney/detect-retina-damage-from-oct-images

THANK YOU !