Question 1.1

In a Multi-layer feedforward neural network with L layers, each layer l with n_l neurons. each neuron has output:

$a_i(l) = h(z_i(l))$, where $z_i(l) = \sum_{j=1}^{n_l-1}[w_{i,j}a_j(l-1) + b_i(l)], i = 1, \ldots, n_l, l = 2, \ldots, L$ .....(1)

where h is an activation function that can be (relu, sigmoid, softmax, linear, tanh, swish,...).

As information enters the network and propagates forward, we eventually get an output $a(L)$ at the final layer of the network. And so measuring the error of the network's prediction w.r.t the known real target values r for input x, using the mean squared error is given by the following:

$E = \frac{1}{2}||r - a(L)||_2^2$ .....(2)

Since machine learning is essentially just optimization, we use the above error as our objective function to be minimized, by changing the network parameters in a smart way. This great tool is backpropagation of the error signal to all the layers of the network to inform the weights by how much they need to change in order for the loss to decrease. Since E is a compound function in z as well, we make use of the chain rule to calculate the derivative of the loss w.r.t the weights for the hidden layers, starting with the final layer and going back:

$\frac{\partial E}{\partial z_j(L)} = \frac{\partial E}{\partial a_j(L)} \frac{\partial a_j(L)}{\partial z_j(L)}$

with: $\frac{\partial E}{\partial a_j(L)} = -2 \times \frac{1}{2}(r_j - a_j(L)) = a_j(L) - r_j$ using the MSE loss

and: $\frac{\partial a_j(L)}{\partial z_j(L)} = h(z_j(L))(1 - h(z_j(L)))$ using the sigmoid activation in the last layer, and so we have that

$\frac{\partial E}{\partial z_j(L)} = h(z_j(L))(1 - h(z_j(L)))(a_j(L) - r_j)$ .....(3)

this only gives us the formula for derivatives in the final layer, to find the derivatives of the loss in the second last layer of the network, we have

$\frac{\partial E}{\partial z_j(L-1)} = \sum_i^{n_L} \frac{\partial E}{\partial a_i(L)} \frac{\partial a_i(L)}{\partial z_i(L)} \frac{\partial z_i(L)}{\partial a_j(L-1)} \frac{\partial a_j(L-1)}{\partial z_j(L-1)} = \frac{\partial E}{\partial z_i(L)} \frac{\partial z_i(L)}{\partial a_j(L-1)} \frac{\partial a_j(L-1)}{\partial z_j(L-1)}$

where the sum i is over all n_L neurons in the layer infront. this provides us a general formular for any layer l to have:

$\frac{\partial E}{\partial z_j(l)} = \sum_i^{n_{l+1}} \frac{\partial E}{\partial z_i(l+1)} \frac{\partial z_i(l+1)}{\partial a_j(l)} \frac{\partial a_j(l)}{\partial z_j(l)} = \frac{\partial h(z_j(l))}{\partial z_j(l)} \sum_i^{n_{l+1}} \frac{\partial E}{\partial z_i(l+1)} \frac{\partial z_i(l+1)}{\partial a_j(l)}$

(*we can take the parts that do not depend on i out of the summation*)

from (1) above we get that $\frac{\partial z_i(l+1)}{\partial a_i(l)} = w_{i,j}(l+1)$ therefor we get :

$\frac{\partial E}{\partial z_j(l)} = \frac{\partial h(z_j(l))}{\partial z_j(l)} \sum_i^{n_{l+1}} w_{i,j}(l+1) \frac{\partial E}{\partial z_i(l+1)}$ .....(4)

Now finally, we can express the weight and biases interms or the chainrule derivation above by going one step into the nested function z given in (1) as follows:

$\frac{\partial E}{\partial w_{i,j}(l)} = \frac{\partial E}{\partial z_i(l)} \frac{\partial z_i(l)}{\partial w_{i,j}(l)} = a_j(l-1) \frac{\partial E}{\partial z_i(l)}$

and :

$\frac{\partial E}{\partial b_i(l)} = \frac{\partial E}{\partial z_i(l)} \frac{\partial z_i(l)}{\partial b_i(l)} = 1 \times \frac{\partial E}{\partial z_i(l)}$

where $times \frac{\partial E}{\partial z_i(l)}$ is given by the formula in (4), and the initial value is in the final layer in (3)

Question 1.2

$X = \begin{bmatrix} x_1 & x_1 & \dots & x_{1,n_p} \end{bmatrix} \in R^{n_l \times 1}$

$$W_i = \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,j} \\ w_{2,1} & w_{1,2} & \dots & w_{1,j} \\ . & . & \dots & . \\ . & . & \dots & . \\ w_{l,1} & w_{i,2} & \dots & w_{l,j} \end{bmatrix}_k$$

$a(l) = \begin{bmatrix} h(z_1(l)) & h(z_2(l)) & \dots & h(z_{n_l}(l)) \end{bmatrix}^T \in R^{1 \times n_p}$

where x_k is has dimensionality n for each pattern.

The MSE error is given by:

$E = \sum_k^{n_{p-1}} [r_k - a_k(L)]^2$ where r is the target vector.

From equation (19) and (22) of the notes we get that equation 21 can be expressed in matrix form as :

$\frac{\partial E}{\partial z_k(L)} = \sum_{k=1}^{n_{p-1}} [\frac{\partial E}{\partial a_k(L)} \frac{\partial a_k(L)}{\partial z_k(L)}]$

and by (22) we know for the hidden layer that

$\frac{\partial E}{\partial a_k(L)} = \begin{bmatrix} \frac{\partial E}{\partial a_1(L)} & \frac{\partial E}{\partial a_1(L)} & \dots & \frac{\partial E}{\partial a_{n_p}(L)} \end{bmatrix}_k^T$

now finally since we have shown in Q1.1 that each layers gradients can be expressed as a functionof the layer infront of it

$\frac{\partial E}{\partial z_k(l)} = \sum_{k=1}^{n_p} w_k(l+1) \frac{\partial E}{\partial z_k(l+1)} \cdot \frac{\partial h(z_k(l))}{\partial z_k(l)}$

Question 2.A

I ran out of time for this question.

however, high level, we get the dot product of the image and the filters(the dot product in matrix form or summation in element) and apply the convolution operator...

Question 2.B

How to run the code

To make running my code easy and generally follow the style mosetly used in the open source community, my code is in 4 files that sit in the same folder. 3 are just utility functions.:

1. **fina_project.ipynb - main code that imports the utilities and will run and show everything.**
2. augment.py - utility class that execused the rotations and adds them to training data in a certain way.
3. data_gen.py - data generator for my neural network, it also performs some transformations on the image.
4. visualization.py - utility for visualization of the learned convolution filters.

To be able to run this, I suggest cloning my repository from github (https://github.com/DeepsMoseli/EE616_final_project) and firing up the fina_project.ipynb. This is to ensure all paths are in place. *note only the original* 119 *images are in the repo :)*

Question 2.C

For the task of dental radiography classification , I tried 2 CNN architectures of different sizes to finally get to my final solution. I also explored different ways of augmentation and their merits and my concerns with them, I will explains my final data preparation, my initial model, and my final submission and why I made this choice.

Data preperation and Augmentation.

- I first crop the image from (748,512) to (748,500) to remove the image information.
- I do not resize the images beyond this as I can not visibly see the desease so I have no prio to say the indicator is in the middle or edges of the image.
- Rotation of images was perfomed only on the 119 training images to grow the training dataset, and not on the 20 test images.
- While there was a need to perform rotations $[90, 180, 270]$ to increase the data, I noticed since the images are not square, this introduces black spaces around the image, I suspect this will add extra complexity in learning. I eventually only peformed $[0, 180]$ augmentation to increase and duplicate the minority class instead of non vertical rotation.
- Also on ploting the images, some looked like they are actully rgb and some not, they had different colors in PIL, for example image 10 and image 110 i then applied a grayscale transform.
- 88% of the original training data, and 85% of the test data is class 1, this is a huge imbalance in the classes, especially as the data is already so small.
- I decided to rotate class 1 images with probability 0.3, but always augment class 0 images. This was to try and close the class imbalance while increasing the data.
- To metigate the network just remembering the rotated images, I used a gaussian blur filter on the images, the radius values of the blur were randomly sampled from $U(0, 4)$.
- The data generator simply creates a pipeline from images to our model and applies a few transformations such as normalization. I cannot crop since I dont know where the objects that show disease sit.

Deep learning Models

- #### A 4 CNN + 4 pooling + 1 FC I first put together my own simple neural network with alternating convolution and pooling layers and a final fully connected layer and softmax layer. This network helped me in fully understanding the math of convolutional dimensions. While I could have continued and explored how to add more sophistication to my model such as skip connections and so on, I had a much stronger feeling the realproblem lies in the data and not so much in a golden bullet model. So I quickly decided on using transfer learning that I detail in the next section. This would allow me to spend more time in figuring the data. Below is the model structure for my own implementation.

I use relu after each pooling layer and the fully connected layes and my final output is a softmax.(not the fully connected)

This model failed to learn anything useful for a long time and changing of the training configurations such as learning rate, optimizer, regularizer, number of fully connected layers. It always seemed to just overfit the training data and then predict everything as class 1 in the test dataset, giving an accuracy of 85%, which is misleading. A better metric such as the ROC AUC score would reveal that this model isnt doing very well. and infact simply looking at the confusion matrix shows this. it has a 100 false positive rate. I will now discuss the next architecture I used.

- #### Transfer learning from Pretrained Resnet34 (https://pytorch.org/hub/pytorch_vision_resnet/) Due the limited training data we hace and the high dimensionality, the classification problem at hand is hard to solve by training a complex convolutional network with a large number of training parameters. We simply do not have enough data to learn salient features from the data present. To go around this, I use a pretrained convolutional network. This method involves using a network with learned (instead of random state) weights that represent salient features about images learned from a much larger dataset.

I decided to use the 34 layer ResNet model, trained on the ImageNet dataset and has weights stored in the pytorch hub that i can download. Resnest was introduced in the paper "Deep Residual Learning for Image Recognition" (https://arxiv.org/abs/1512.03385). I replaced the fully connected layer with a fullu connected with relu activation on 8 neurons and

added a final softmax layer for the 2 classes. The pytorch pretrained models expects input images pf shape $(3 \times H \times W)$ where H and W are expected to be atleast 224. It does not requirethat we have square images.

- In training this network, I used the ADAM optimizer with the default learning rate of 0.01 and zero weight decay.
- I played around with the dropout regularizer but found that it only delays the overfittting, the model trained over 50 epochs still resulted missclassification of class 1.
- I also work on using weights for the loss to put more enphasis on class 0 but this also didnt help
- I also considered freezing and unfreezing different parts of the pretrained model such as the final cnn layer, while this looked promising initially, the same thing happend at evaluation.

For my final answer, I can say, without penalizing the model very strictly, it quickly converges on the training set within only a few epochs resulting in 85% accuracy, restricting it a bit leads to a delayed overfitting and still same reasult. using a very high tropup kills it entirely so that it cannot predict even the majority class well. I do not know for sure if being very strict and training longer would have gotten me the 100% on evaluation results or not.