Determination of metastasis of cancer cells

PA 3 – CSE 190

Xiyuan Chen
Student, UC San Diego
xic145@ucsd.edu

Nikhilesh Sankaranarayanan
Student, UC San Diego
nsankara@ucsd.edu

Abstract

This paper describes techniques to train a convolutional neural network to determine whether cancer cells have metastasized in lymph nodes by anaylysing a large collection of images of biopsied lymph nodes. The network trains itself on a random subset of the total dataset and tests itself against the unseen data.

Introduction

Cancer is responsible for thousands of deaths in the USA every year, with breast cancer being the most prolific type, with over a quarter of a million diagnosis each year. This is a serious problem, but with timely detection and treatment, the number of fatalities can be greatly reduced. To aid detection and treatment of the disease, biopsied slides of lymph node sections have to be analyzed to identify metastasis – however, this is a difficult and time-consuming process for pathologists. This paper describes an approach to the challenge by designing a Convolutional Neural Network capable of reading in the large dataset of lymph node whole-slide sections provided by Camelyon16. The network will have multiple layers and varying activation functions through each layer, and produces a spread of likelihoods of having one of 14 defects in each stained slide. The network is described in more detail under the Methods section.

Related Work

This network was inspired by previous neural networks designed by the authors, and the discussions in lectures of Dr. Garrison Cottrell, specifically the tips on constructing the layer architecture and initialization of weights and biases.

Methods

First, the input data has to be classified. Each lymph node scan is labelled depending on what diseases have been identified in it, from a total of 14 possible conditions. Therefore each label describing an image is represented by a list with 14 binary values, a 1 representing the presence of a defect and 0, the absence.

Input and Preprocessing

The dataset originally contains 112120 images corresponding to biopsies of lymph nodes. Each image is accompanied by a label that describes what condition the patient has, all contained in a .csv file. These input images are pre-processed by scaling down to size 256px*256px, with 3 color channels. This significantly improves processing speed while preserving important features in each image. The Network reads in each label from the csv file into a Pandas DataFrame, and translates it into a list with 14 values, and creates a Tensor of all the labels. Then, each image in the scaled-down dataset corresponding to a label is read in as a numpy array and converted to a Tensor. Now, both images and their labels are contained within Float Tensors.

Architecture

The convolutional network has the following architecture for each of the hidden layers:

- Input layer
- 4 convolution layers
- 3 linear activation function layers
- 1 sigmoid layer

Each layer undergoes batch normalization after its output is produced to ensure more accurate weight updates. The starting weights for each layer are initialized by Xavier Initialization. This was applied using the built-in function provided by the PyTorch library. Bias term is set to a small negative number (-0.1). After each convolution layer, we apply a ReLU function to the output, then batch normalize it. Then, it is subsampled using the max_pool2d function. The final layer has a Sigmoid activation function. The reason for this is to produce a probabilistic output i.e. the chance that our network believes a particular disease is identified in the input image.

The breakdown of the network is as follows:

Input: Takes 3x256x256 inputs as a Tensor (the images).

First layer - Convolutional layer. Filter size: 5x5. Weights are 3x6x5x5. Output/Activation: 6x252x252. Max pool: 2x2. Downsampled to: 6x126x126.

Second layer - Convolutional layer. Filter size: 7x7. Weights are 6x16x7x7. Output/Activation: 16x120x120. Max pool: 2x2. Downsampled to: 16x60x60.

Third layer - Convolutional layer. Filter size: 5x5. Weights are 16x24x5x5.

Output/Activation: 24x56x56. Max pool: 4x4. Downsampled to: 24x14x14.

Fourth layer - Convolutional layer. Filter size: 5x5. Weights are 24x40x5x5. Output/Activation: 40x10x10. Max pool: 2x2. Downsampled to: 40x5x5.

Fifth layer – Linear layer. Weights are (40*5*5)x120. Output/Activation: 120. This is a fully connected layer (no downsampling).

Sixth layer – Linear layer. Weights are 120x84. Output/Activation: 84. Fully connected layer.

Seventh layer – Linear layer. Weights are 84x14. Output/Activation: 14. Fully connected layer.

Eighth layer – Sigmoid layer. Output/Activation: 14. This is the output layer.

Training/Testing

The network is trained on a subset of the total dataset (90%). We use 10-fold cross validation to prevent overfitting to the same dataset. The dataset is split into 10 different slices. One of these is assigned as the test set, and the remaining slices comprise the training set as 9 mini-batches. Each mini-batch is of size 112120/10 = 11,212.

The network is trained on each mini-batch. Once the weights are updated, gradients are set to zero and training begins on the next mini-batch. This is repeated for all 9 mini-batches. Finally, the network is tested against the last 1/10 of the dataset, and the loss is computed.

The training process itself is run 10 times. There is no early stopping criteria. This might seem odd, but for the size of our batches and the few number of epochs our loss decreases quite steadily. We feel that this justifies always completing the full training.

Each epoch, a different selection of 9 slices is chosen as the mini-batches, with the 10^{th} slice being the hold-out set. The best performing network is selected after 10 such iterations with different hold-out sets. This approach was inspired by a desire to shuffle the dataset differently each time and ensure that accuracy was consistent regardless of which slices of data were used for testing/training.

The loss function used is BCELoss from the PyTorch library. The reason was since we are avoiding one-hot encoding as we are producing multiple possible output classes, we could not use standard Cross Entropy loss. The BCELoss function works well because our output targets are binary (i.e. definitely between 0 and 1).

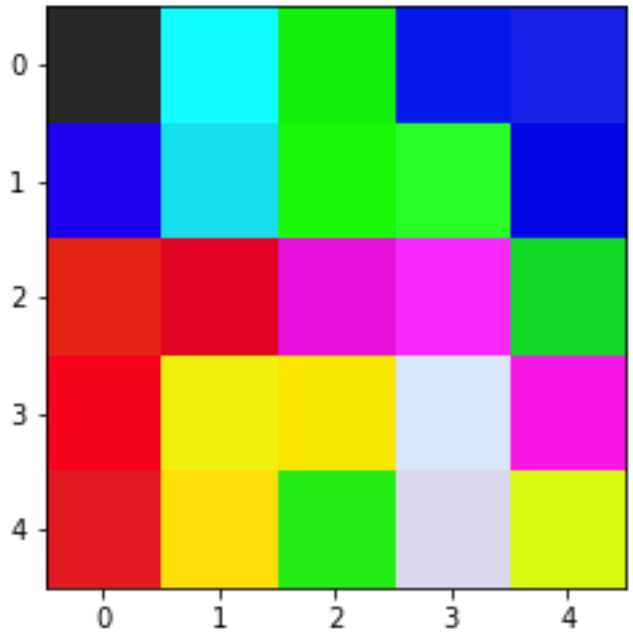
This was used in conjunction with the Adam Optimizer function. After the forward propagation step, the backward() function from PyTorch completes the backpropagation step, generating the output tensor. This is of the same dimensions as the labels tensor, allowing for direct one-to-one comparison of values. The training step continues for all 9 slices of training data.

Once the network has learnt its weights on the training data, the network is run on the test data and the output is compared to the corresponding section of the labels Tensor. Values that are ‘1’ in the labels indicate presence of a particular ailment. Similarly values close to ‘1’ in the output Tensor indicate that the network believes

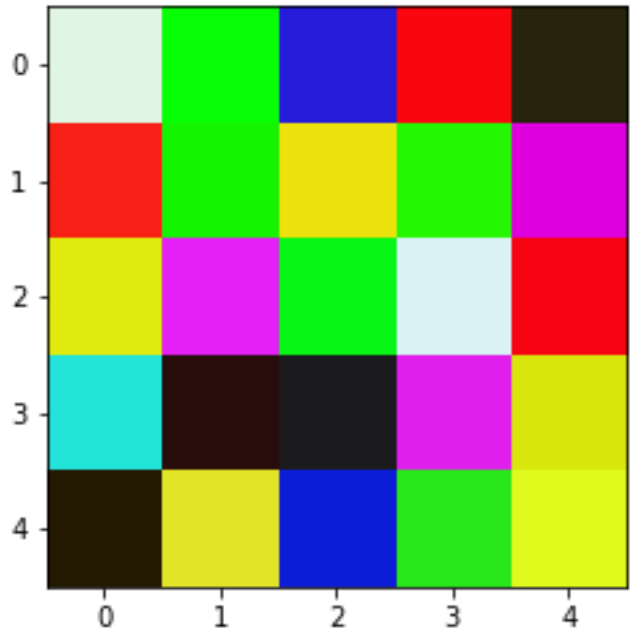
the ailment is more likely to be present. The criterion() function produces a Tensor that maps the difference between expected output and actual output (as a perfectly correct output would perfectly match expected output). This error rate is measured over each epoch for different networks, and all results are compared. Finally, the network with the lowest loss is reported.

Visualization

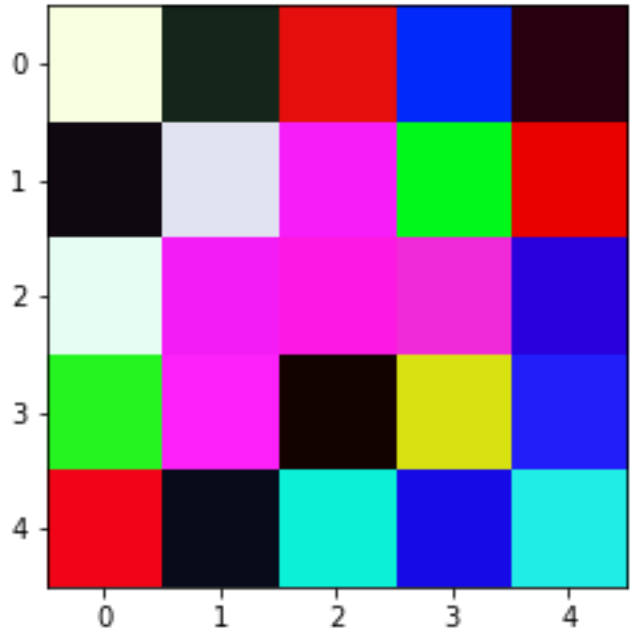
To visualize the filters in the network, we extract the weights of the network using the `.parameters()` method. Since the first layer takes in inputs of 3 channels, we can visualize it as RGB images. There are 6 images that we can visualize and they are shown below:



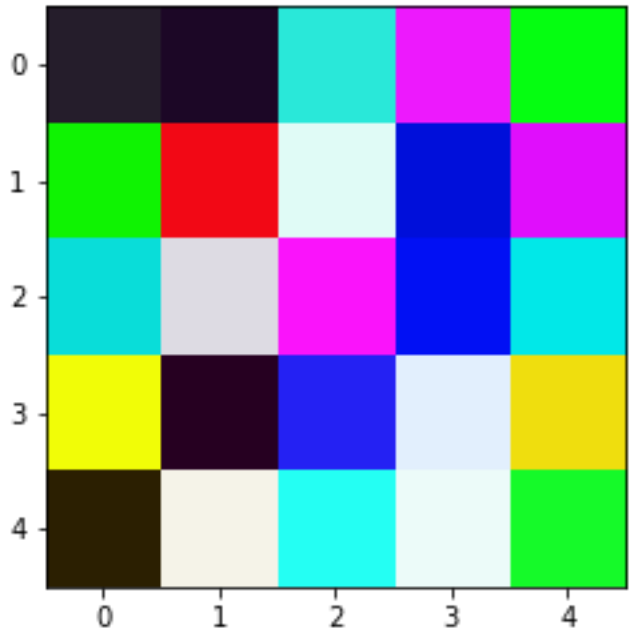
Weight 0 Layer 1



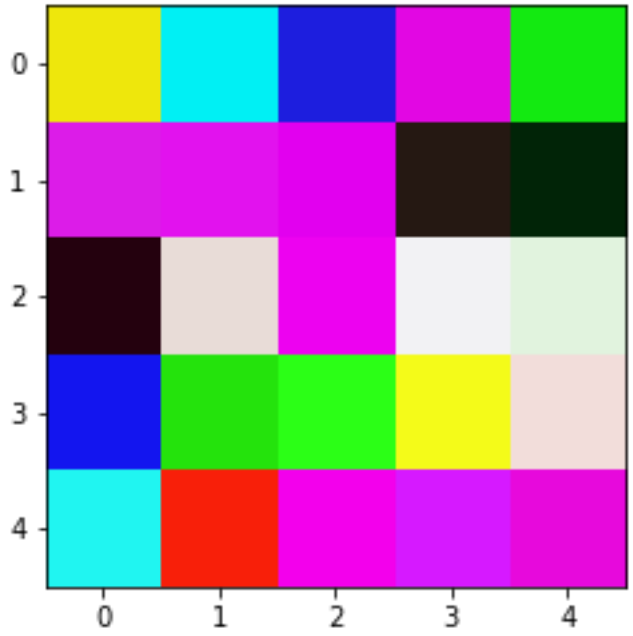
136 Weight 1 Layer 1



137 Weight 3 Layer 1



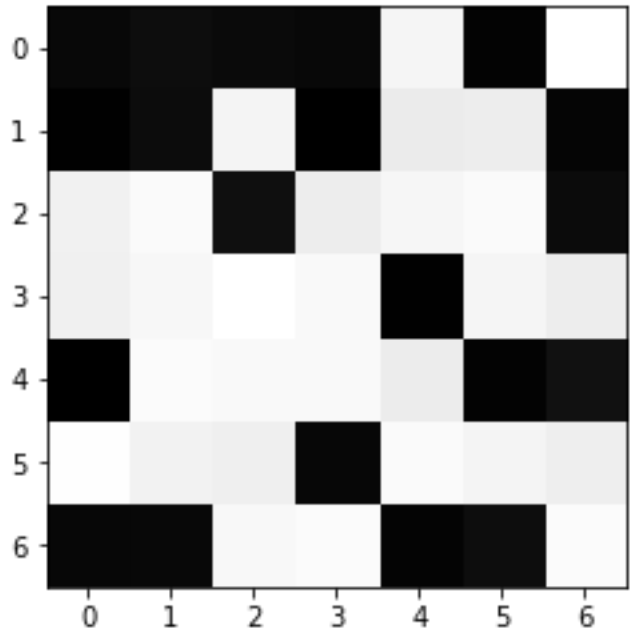
Weight 4 Layer 1



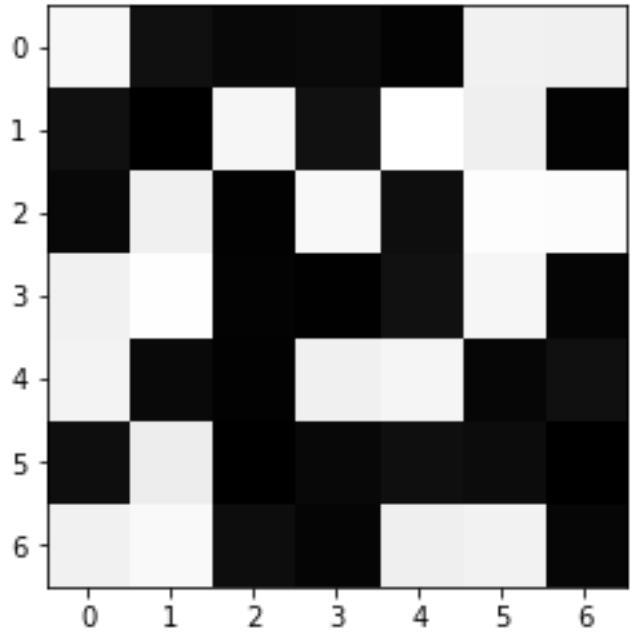
Weight 5 Layer 1

As we can see from the graphs, these images of weight matrices are not very intuitive. One cannot simply tell what feature that filter is extracting.

As for inner layers, we can only turn them into grayscale images. We will only show 2 of them from the second layer, since they also do not make any (intuitive) sense to us.



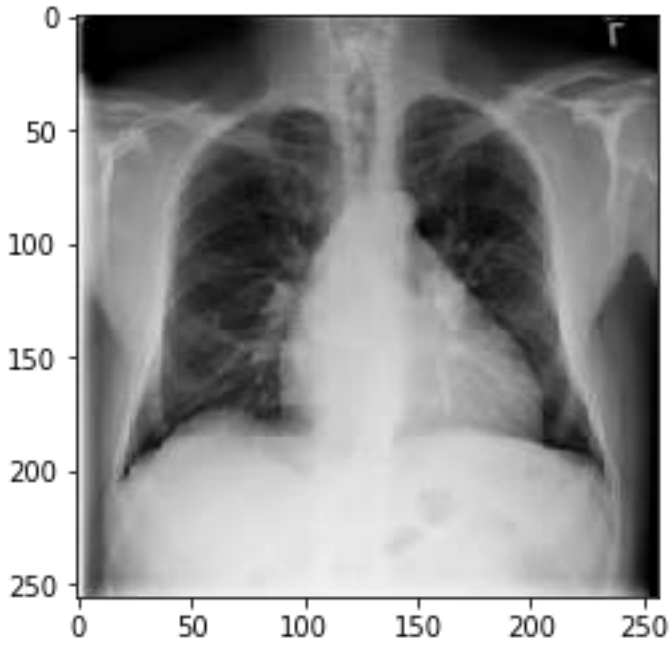
Weight 0 Layer 2



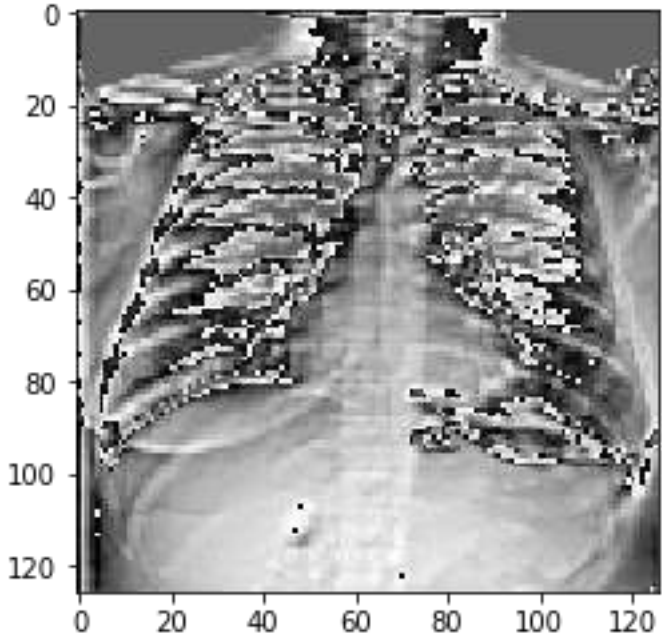
Weight 1 Layer 2

As you can see from the images of weights in the first and second layers, the features are not clearly identified. They are much clearer if we apply these weights to the inputs and visualize the activation map. The activation of the first image through the first 3 layers is shown below. We are only showing the first 6 activations of each layer since there are too many.

Original image:

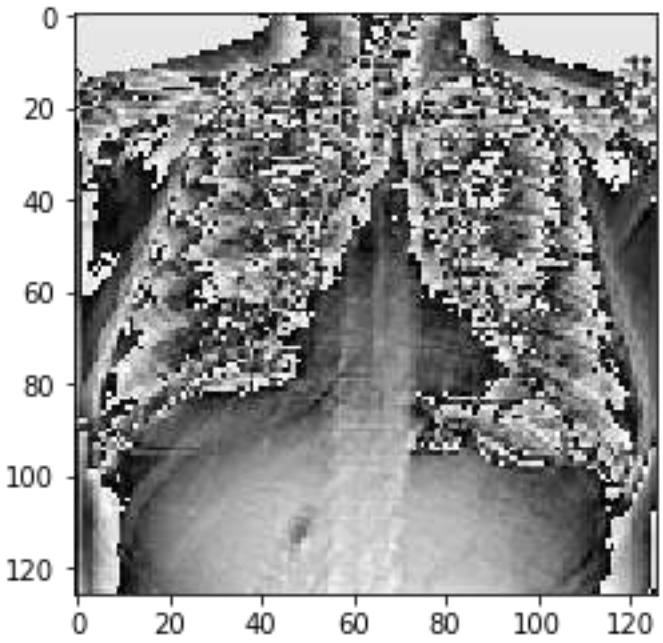


First layer:



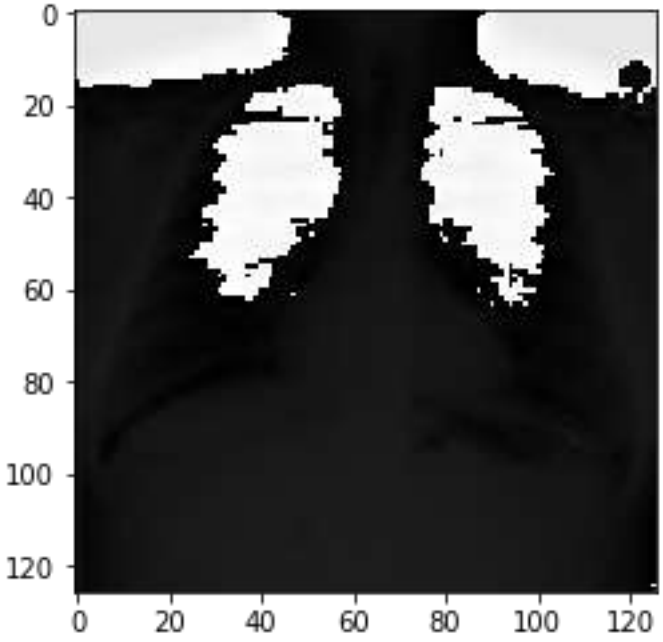
Activation 0

A possible explanation for this filter is that it detects the bones and edges between black and white.



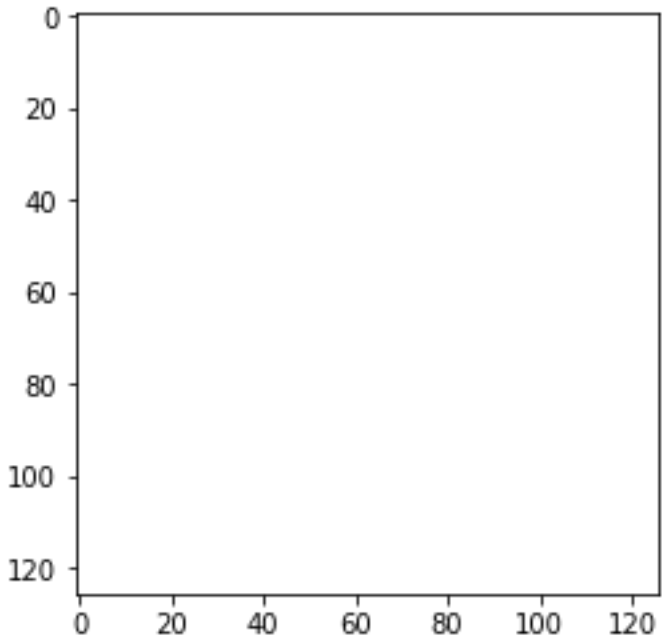
Activation 1

A possible explanation for this filter is that it defines countering from black to white as one can see that the black and white pixels are reversed.



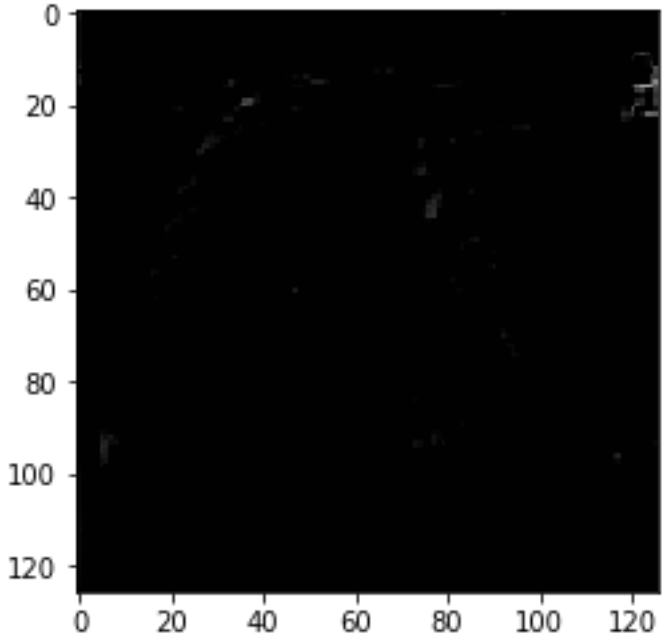
Activation 2

Here we can see that the lung is highlighted by white and other things except for the background is black. This filter is probably extracting the shape of the central region of the lung.



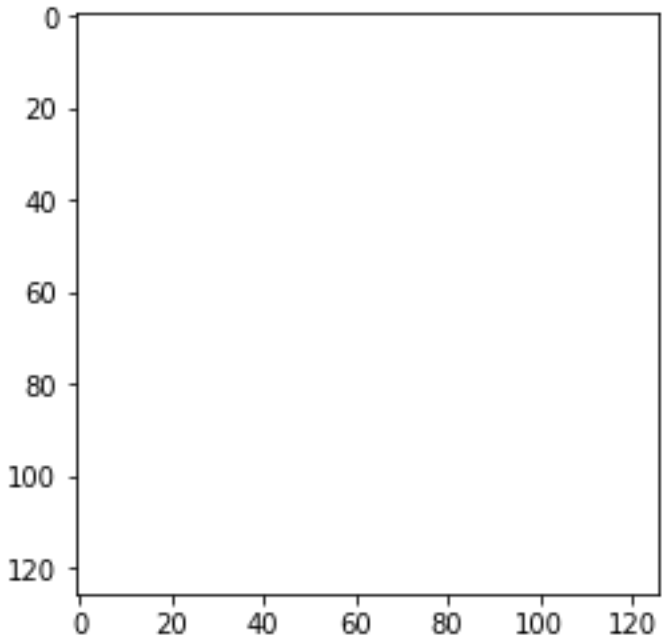
Activation 3

An interesting activation map because it's all white, and it suggests that this image does not contain any pattern that matches this filter's feature.



Activation 4

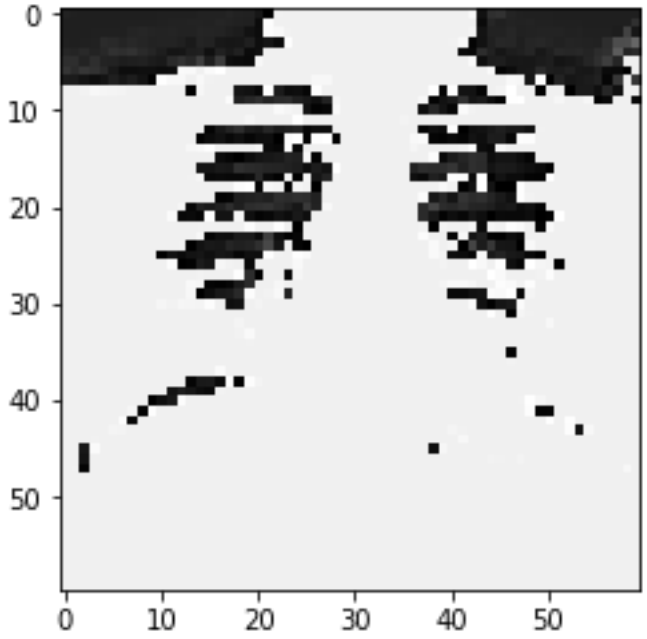
Same as the last one and a reasonable guess is that this filter is the countering of the last filter.



Activation 5

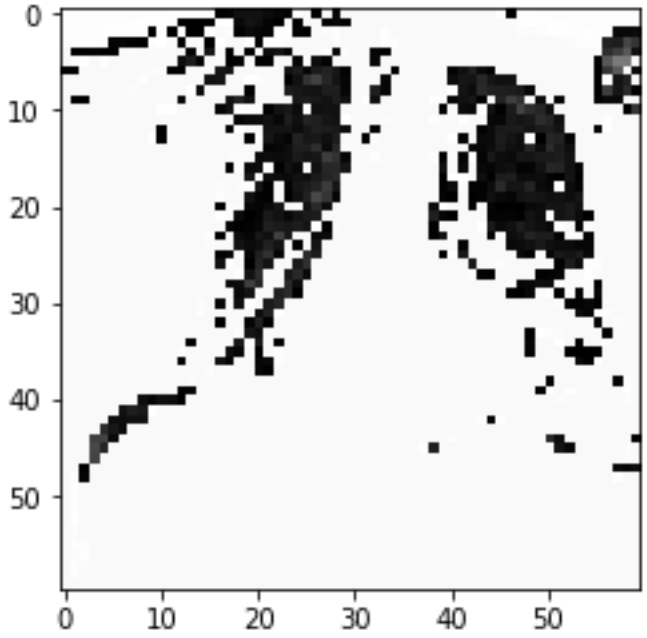
Same as activation map 3. We don't know what feature it is extracting.

Second layer:

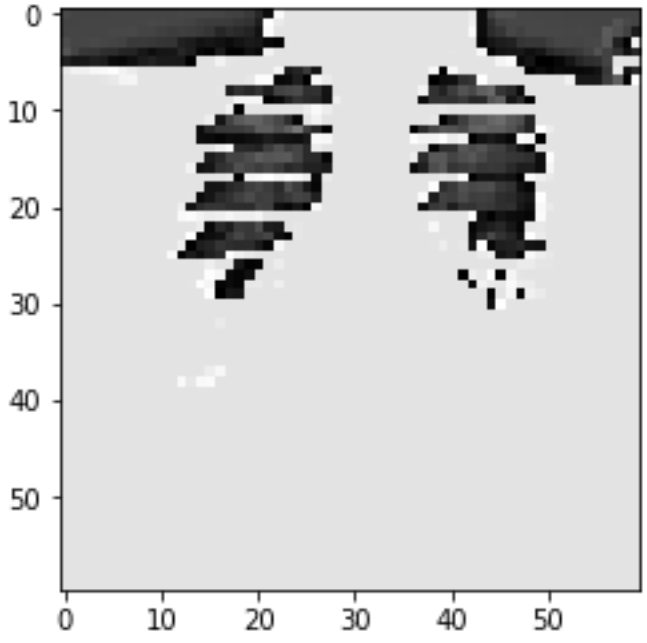


Activation 0

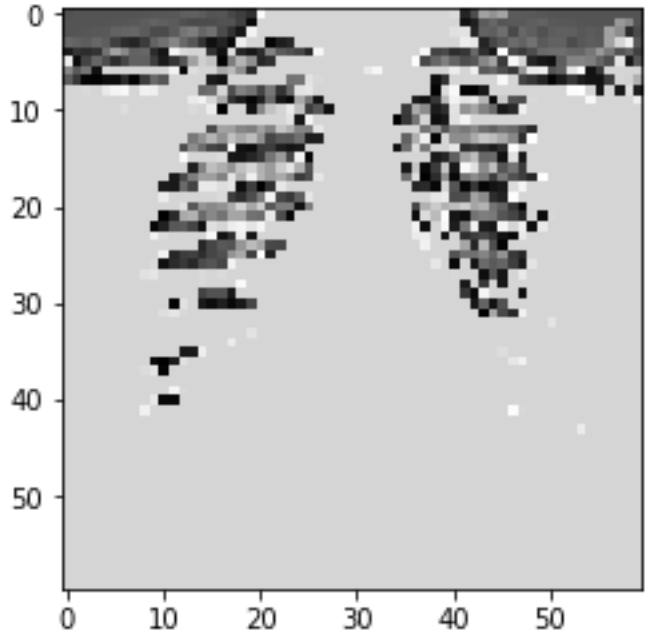
In the second layer, we can see that after the down sampling, the image is a little abstract but we can still see the shape of the lung.



181 Activation 1

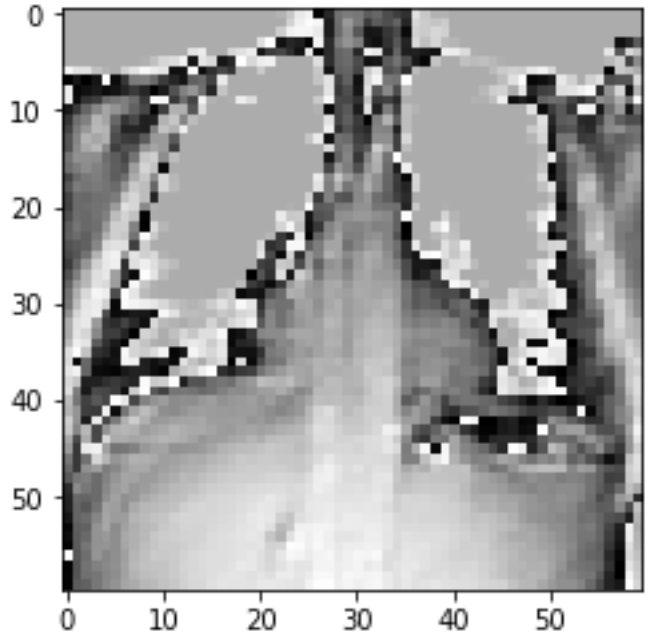


182 Activation 2



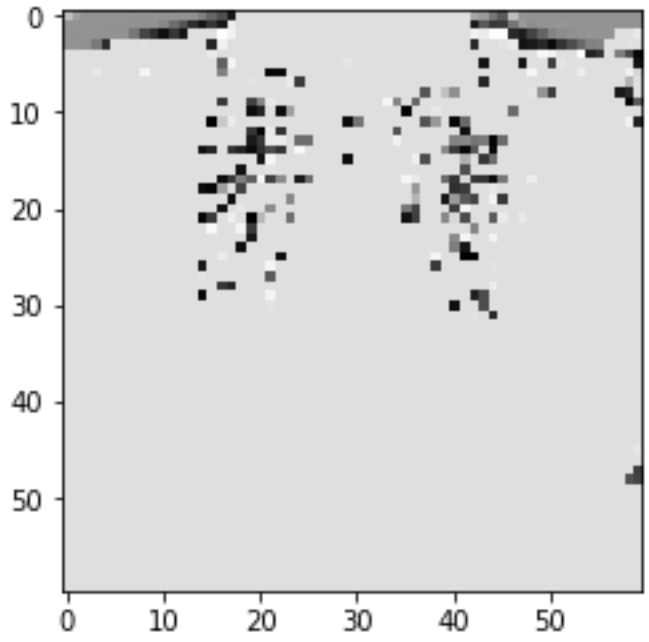
Activation 3

It’s harder for us to determine what feature this unit is extracting at this point but it is still isolating the lung cavity.



Activation 4

This activation is very different from the previous ones and it’s probably extracting the other sections besides the lung.

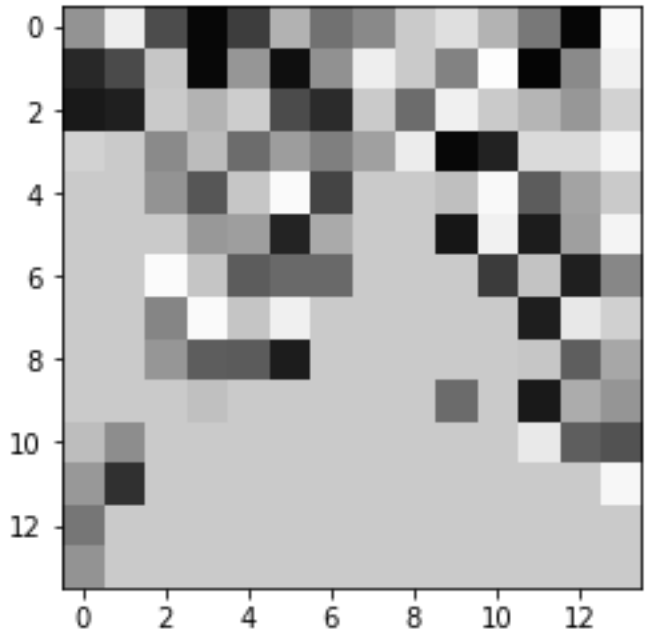


Activation 5

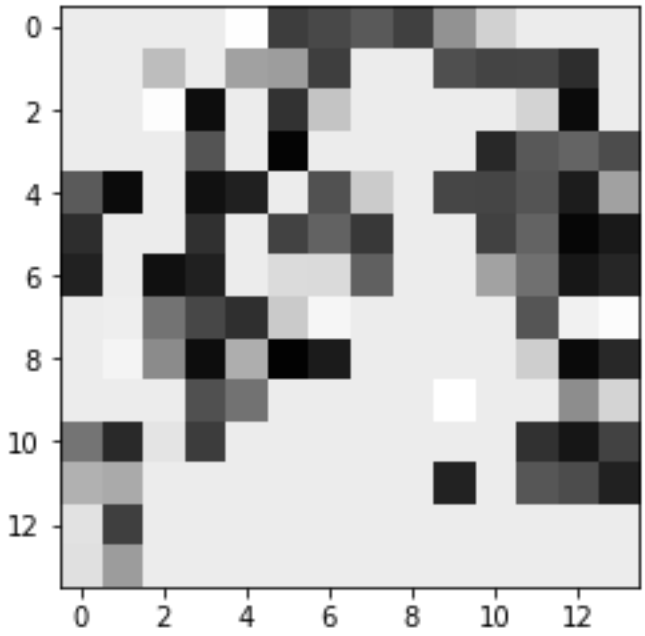
We have little idea of what this represents other than random dots representing the lung.

Third layer:

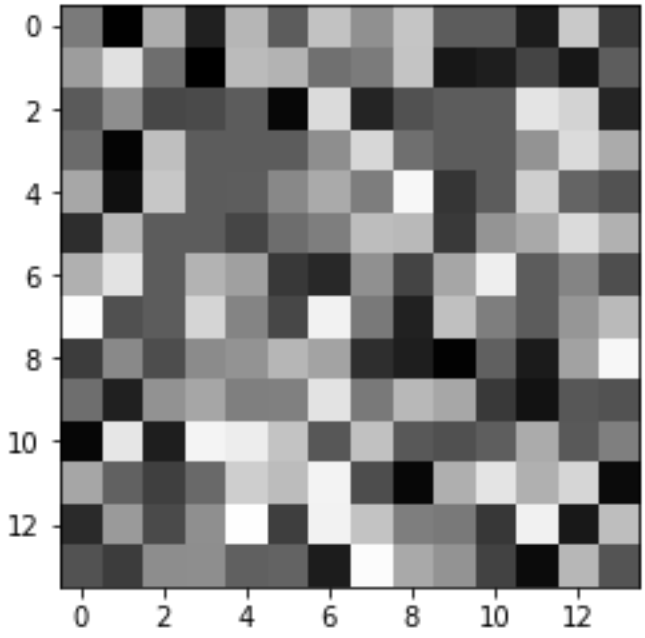
Here in the third layer, everything becomes so hard to interpret. We list out first 6 activations here and they are all very abstract.



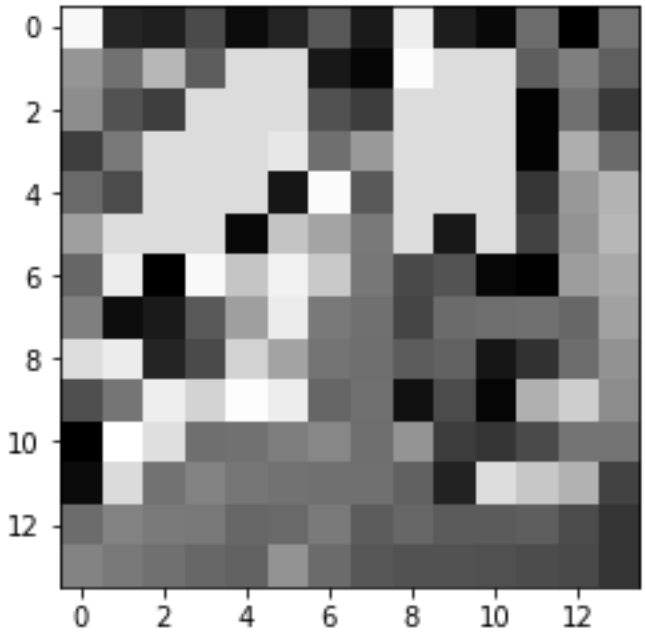
Activation 0



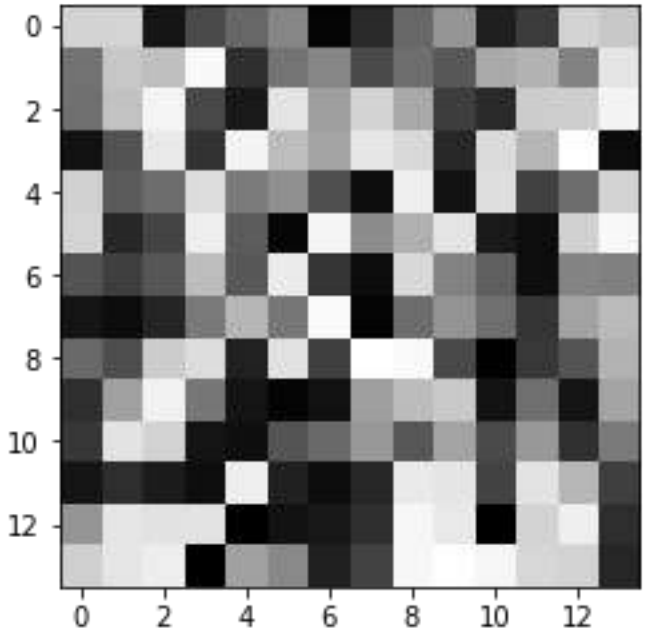
196 Activation 1



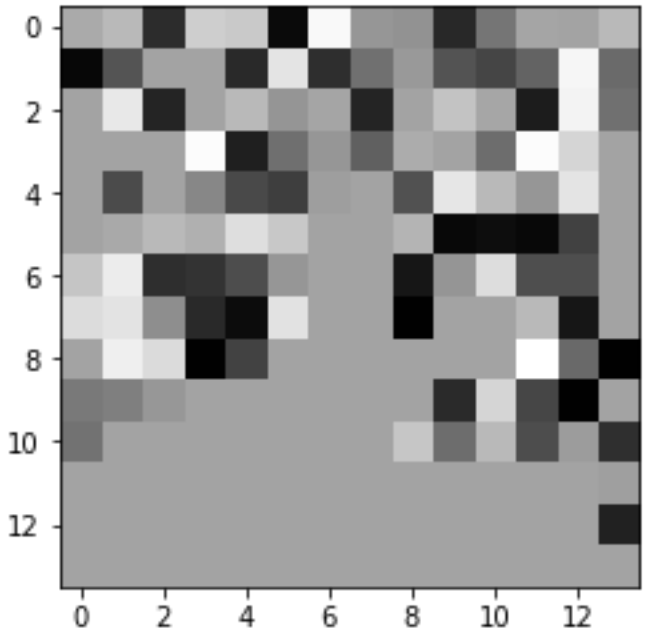
197 Activation 2



198 Activation 3



199 Activation 4



Activation 5

The vague shape of the rib-cage can be made out in a couple of images but by now the features have passed into the machine's control, and we cannot derive any intuitive meaning from it. Further layers only increase the level of abstraction.

Transferred Learning

We used the pre-trained Microsoft ResNet 18 as our base network. We removed the last fully connected layer and replaced it with a fully connected Sigmoid layer. We freeze every other layers' weights and train the network to update only on the last layer. The plot of loss for that layer is shown below:

This was run over 25 epochs on the same batch size and 10-crossfold validation. The results are documented in the next section.

Results

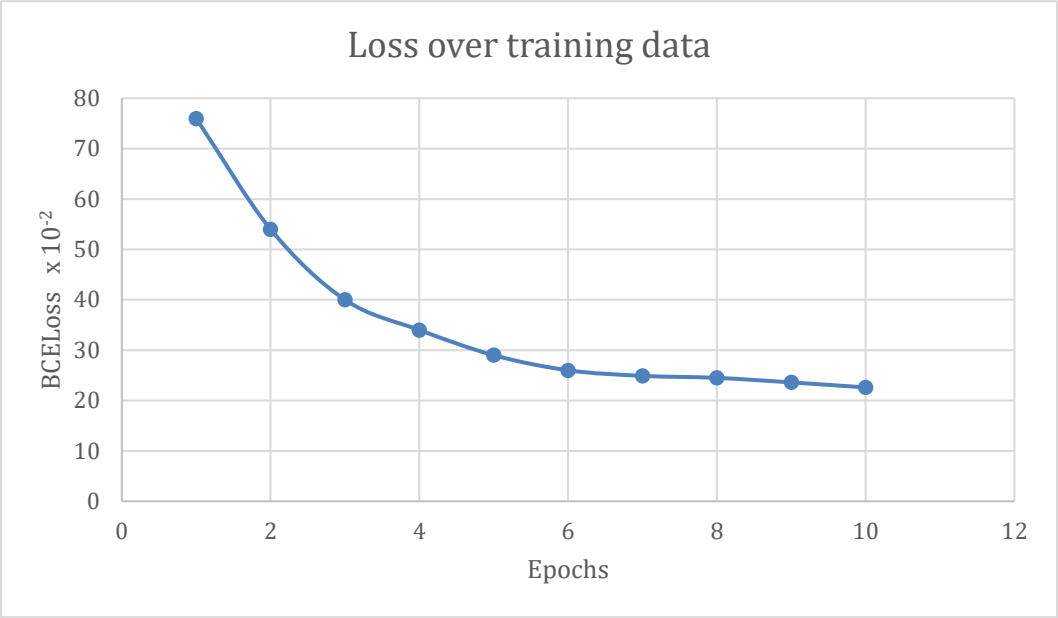
The network performs adequately on the dataset. General trends observed are decreasing loss and increasing accuracy (also known as Performance) as number of iterations over a mini-batch increases.

Loss for our network

Below is a plot of loss over training data, while training over each mini-batch. The loss is the BCELoss of the output for each minibatch of the input.

Note: For the sake of time when reporting, a subset of 500 images was used but

the same trends are seen over the whole dataset. Larger subsets were randomly prone to running out of memory on the GPU.



As you can see, the loss steadily decreases as the network continuous training a batch. This is repeated 10 times with different parts of the total dataset being taken as the training set. The best network from these 10 is chosen and final loss comes to around **0.22** after training.

It is notable that every time we run our program, the result is slightly different, and most of the time we can get a loss of only 0.17 from network number 4. We think that this is probably because the training set of net 4 best represents the whole dataset, so it fits the testing set well. In general, the loss is around 0.20.

Performance/Accuracy of our network

In measuring performance of the network, we needed a way to interpret the output of the network. The network produces a tensor that contains in each row a list of 14 values between (0,1) that describes the likelihood of any of 14 diseases identified. Consider one particular image –values closer to 1 in its output list indicate greater chance of a disease. Since multiple output classes can be present, we can not take the maximum value in the list. Instead, we consider the highest values in the list that are sufficiently larger than the remaining values. To this extent we implement and justify our own criteria to determine this.

In each list, we can have multiple possible correct outputs. The number of such outputs itself is a variable between each image. So we consider only the output

values that are larger than the others by a significant margin, but must allow for different number of positive classifications for each image. The Arithmetic Mean of the list is a good indicator of the separation between the high and low values in a list with diverse values (which is what we observed from the output of the training data). By segregating output values around the mean we can produce a predicted output that is of a similar format as the input labels tensor (containing only 0s and 1s).

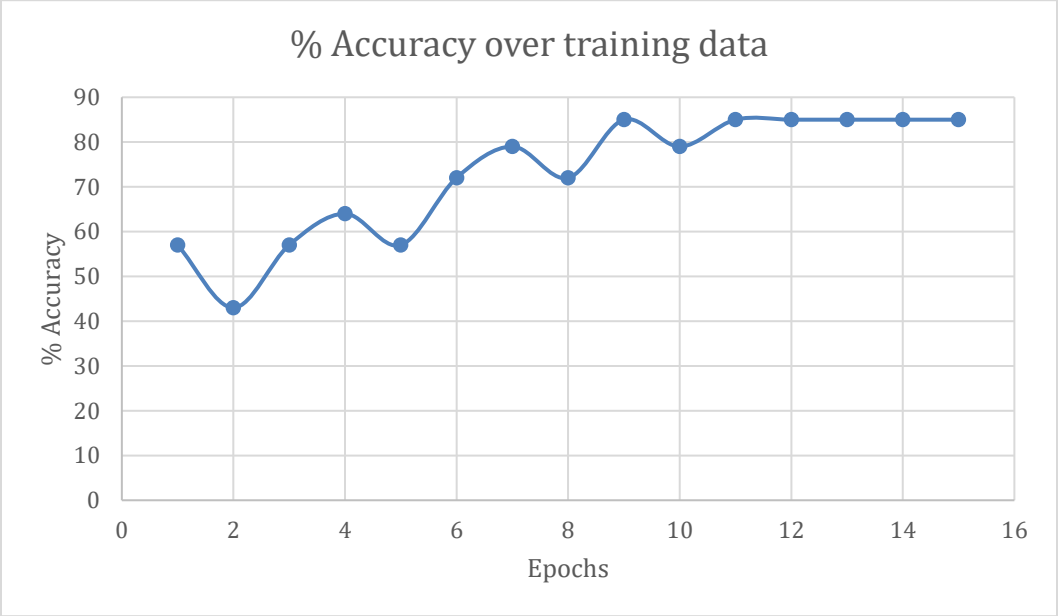
For example: If the network outputs this list for a particular image:

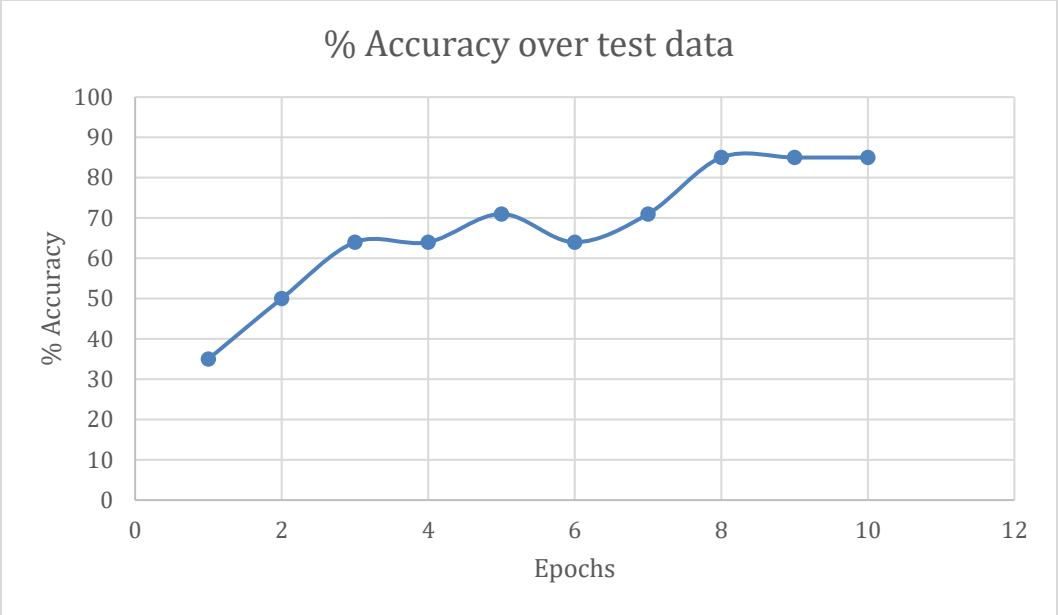
```
[0.62 0.03 0.1 0.21 0.78 0.43 0.12 0.45 0.81 0.43 0.01
0.07 0.68 0.13 ]
```

Then the mean for this list would be **0.34**. Now partitioning it based on this mean, we get the Predicted Output to be:

```
[ 1 0 0 0 1 1 0 1 1 1 0 0 1 0]
```

Using this prediction model, we compute the **Accuracy** of the model by the number of such lists that were correctly classified i.e. that matched the test data. Below is a plot of accuracy vs iterations for the training data.

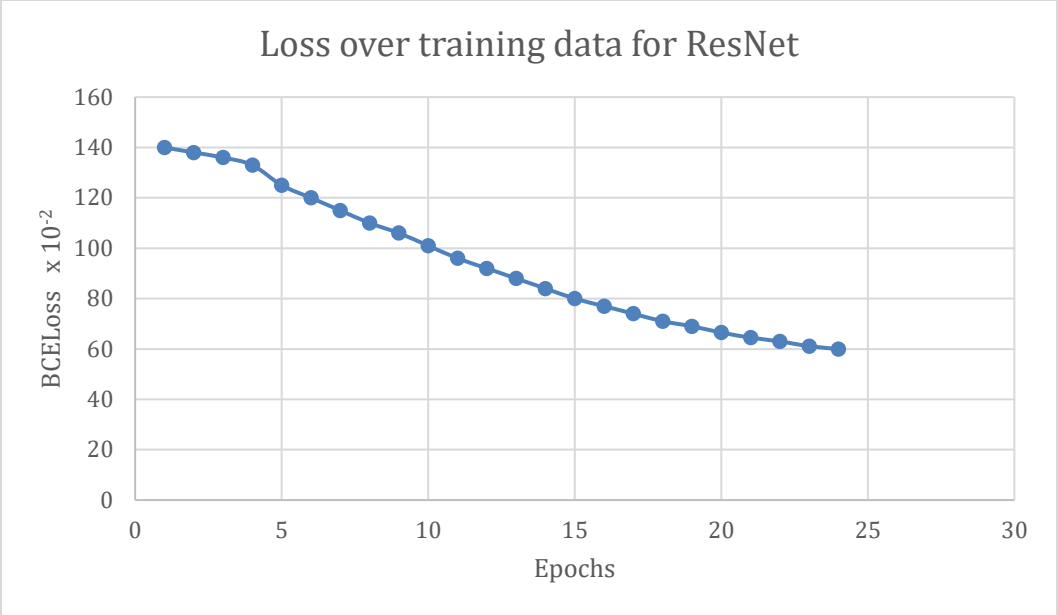




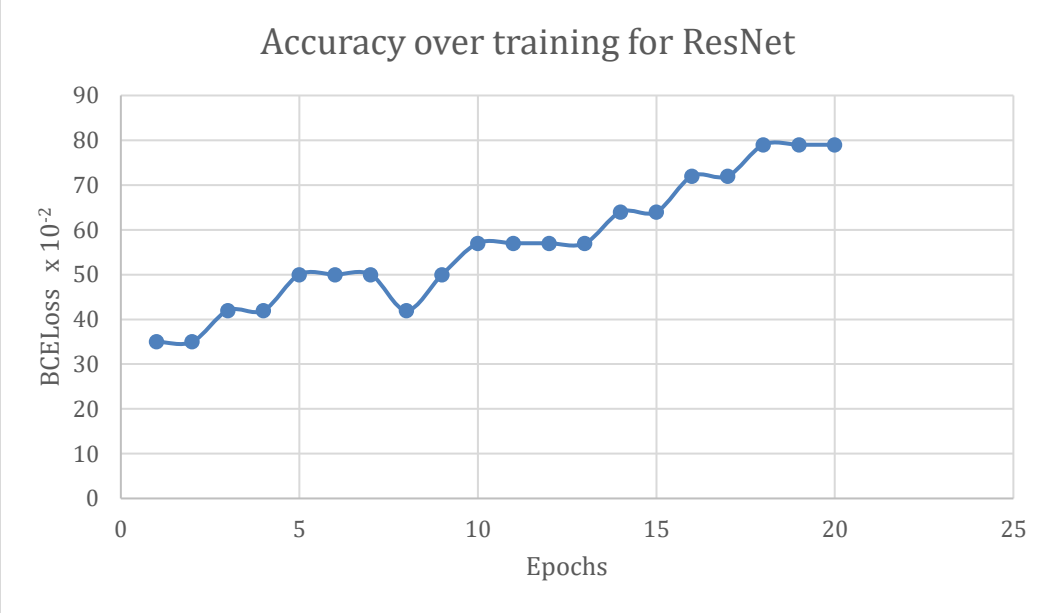
Here we can see that the accuracy is not as good as we expected, because our network can reduce the loss to around only 0.17 and we still ended up with an accuracy below 90% on the test data. We think that's probably because of two reasons. First, the way we generate our prediction is not perfect. If we can look at the distribution and pick the high values of probability manually (or by some algorithm), we can probably do better. Second, because all we have are probabilities and one person can be classified into several classes, it is very possible that we have too many false positives if they are distributed evenly or too many false negatives if one of the probability is extremely high.

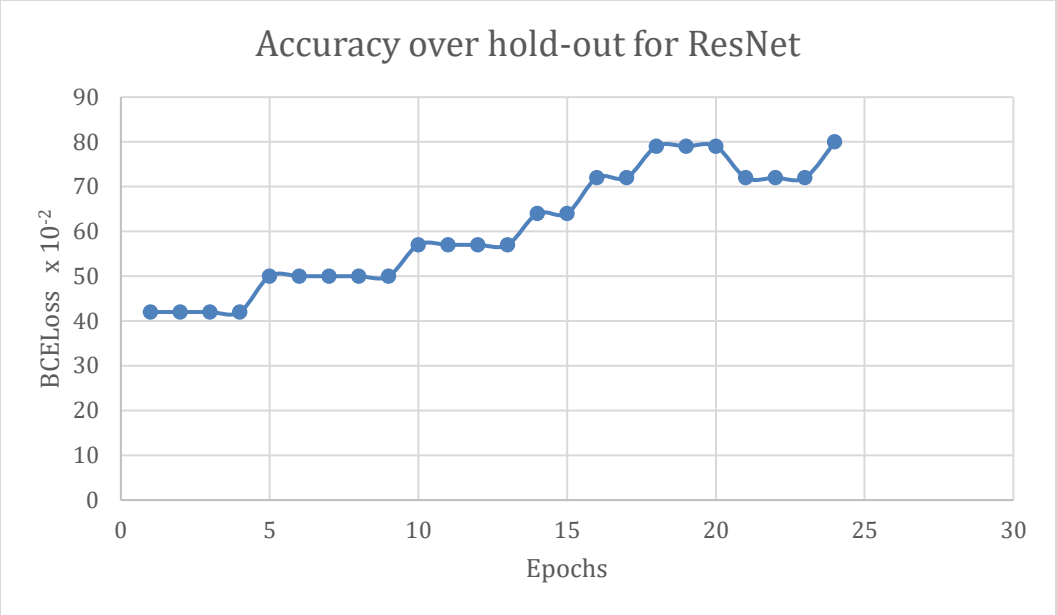
Loss of Transferred Learning network

The results from the Microsoft ResNet are documented below.



The loss function smoothly decreased as shown above for all combinations of training data. It also ran each network in under 5 seconds. We were quite surprised at the speed. Accuracy is similar, though not as high as we would have expected. It is likely that our accuracy function is not well-fitted to the ResNet. The increase in accuracy over epochs is steady, however, and mono-tonic, which is an improvement over our network.





Compared to our version of neural network, we think it is reasonable that ResNet is not reducing as fast because we are freezing the weights of inner layers and we cannot modify our feature extractors and make them learn. ResNet may be good at identifying ImageNet images, but not as good at identifying cancers.

Discussion

The results of this network are quite promising. Despite not achieving very high accuracy, the network still has proven to learn something as it has a better success rate than random guesswork! The methods we chose for normalization and calculating accuracy were unique, and it worked well in this case. The pre-trained network was straightforward to implement and actually outperformed our network in terms of execution time and minimizing loss. This is probably because it had a much larger image dataset to train itself on, and likely has more hidden layers and weights so it can adjust to a new dataset quickly.

Our network also has a good track record though. The decision to shuffle the dataset on each training cycle was instrumental in ensuring that we did not overfit the data to any particular subset of the total dataset, and testing was always done on a blind subset. Added features of Batch normalization made sure that each training cycle was able to update the weights without influence from previous, different updates. Zeroing the gradient helped with this too. Finally, the architecture that was inspired by the pytorch.org tutorials was also helpful in breaking down each input image into the subsamples and identifying the necessary features.

Teammate Contributions

Pair Programming was observed in every stage of the project. The work was split evenly across programming, report writeup and reporting results.

Xiyuan was responsible for deciding the architecture of the convolutional layers and constructing the torch.nn module. He also implemented the iterative step to iterate over all mini-batches of the program. He installed the Microsoft ResNet and modified its output layer to suit our purposes. He produced visualizations of weights and activations of hidden layers in the ConvNet.

Nikhilesh was responsible for implementing loading of images and labels into the program. He also carried out initialization of weights and biases, and the application of modifiers like Batch normalization and Xavier initialization. He implemented the GPU code and handled the data structures for teaching and training tensors. He implemented the code to calculate and plot loss and accuracy, and wrote the Report.

References

- www.pytorch.org/docs
- www.pytorch.org/tutorials
- https://discuss.pytorch.org/
- Lecture slides of Dr. Garrison Cottrell.