Multi-Label Classification of X-Rays using Convolutional Neural Networks

Eric Zhou

Northeastern University

CS4100

December 8, 2020

Abstract

The use of convolutional neural networks as a tool to assist radiologists has the potential to significantly improve hospital efficiency and care quality. Its ability to detect important features in an image makes it a practical tool to use for X-ray analysis. I used two different convolutional network architectures, AlexNet and MobileNet. The use of different loss functions and learning rates was also studied to determine the optimal configuration of the networks. Because it is much more common for an image to be negative for some finding, it was difficult to find a configuration that would result in predictions with high binary and absolute accuracy. Using binary cross entropy as a loss function, a network trained on the NIH Chest X-Ray Dataset achieved a binary accuracy of 0.9 and an absolute accuracy of 0.3.[1] Using a weighted loss function resulted in a binary accuracy of 0.69 and an absolute accuracy of 0.25. The weighted loss function showed promising results after each training epoch and could achieve a higher overall accuracy given more training time.

Introduction

The chest X-ray is the most frequently used medical imaging exam. It is often used as a basic screening study to quickly analyze a patients' general condition and determine case priority. When there are many studies needed to be read, there is no way to determine which to read first. Convolutional neural networks are the most popular architecture used for image classification and can detect important features of an image without human supervision. The X-rays of a patient who may have some critical finding could be buried by the hundreds of other cases in the list. The use of an image classification program as an initial screening process could sort cases

[1] Dataset can be found at https://www.kaggle.com/nih-chest-xrays/data

by their degrees of urgency and improve the patient experience, patient care, and turnaround time.

I decided on using the National Institutes of Health's chest X-ray dataset because it had the largest amount of images to work with. It also contained the findings for each image in a CSV file for easy reference.

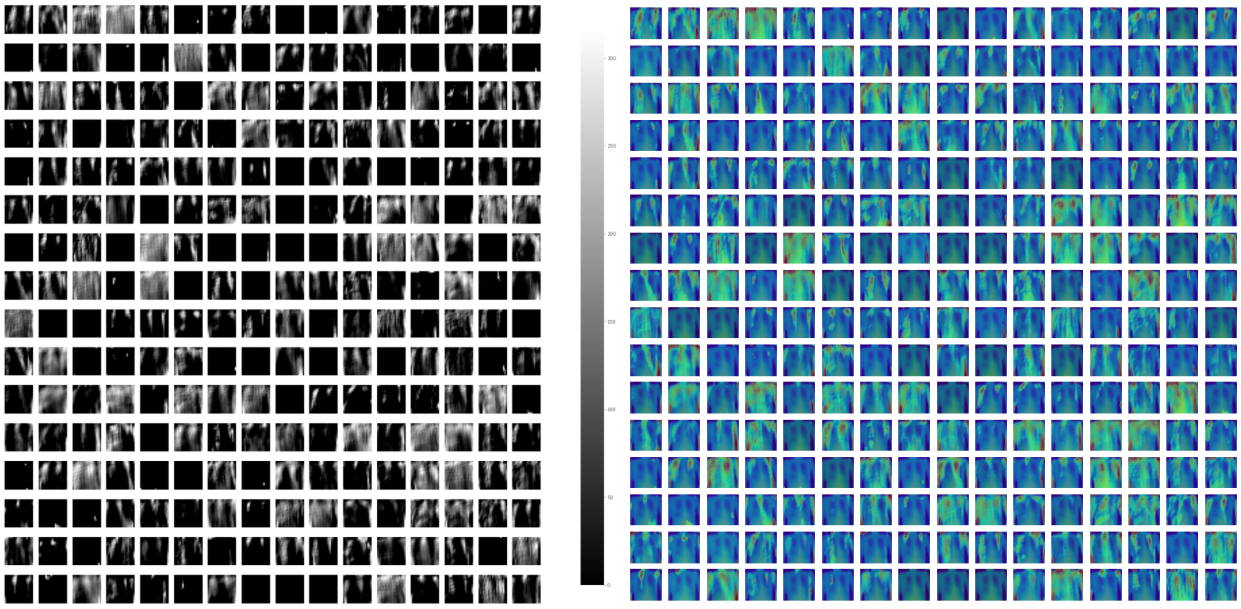
The original dataset has 14 classes of labels. I chose to combine some labels because of how similar they are represented by an X-ray image. Nodule and Mass were combined under the "Nodule" label. Infiltration, Consolidation, and Pneumonia were classified as "Consolidation." For consistency, I removed any lateral views, using only AP and PA views.

Methodology

The models all take in an image input with varying size depending on the architecture. The output first contained an array of 11 values between zero and one, mapped to the finding labels. The values represent the probabilities of the image containing each finding.

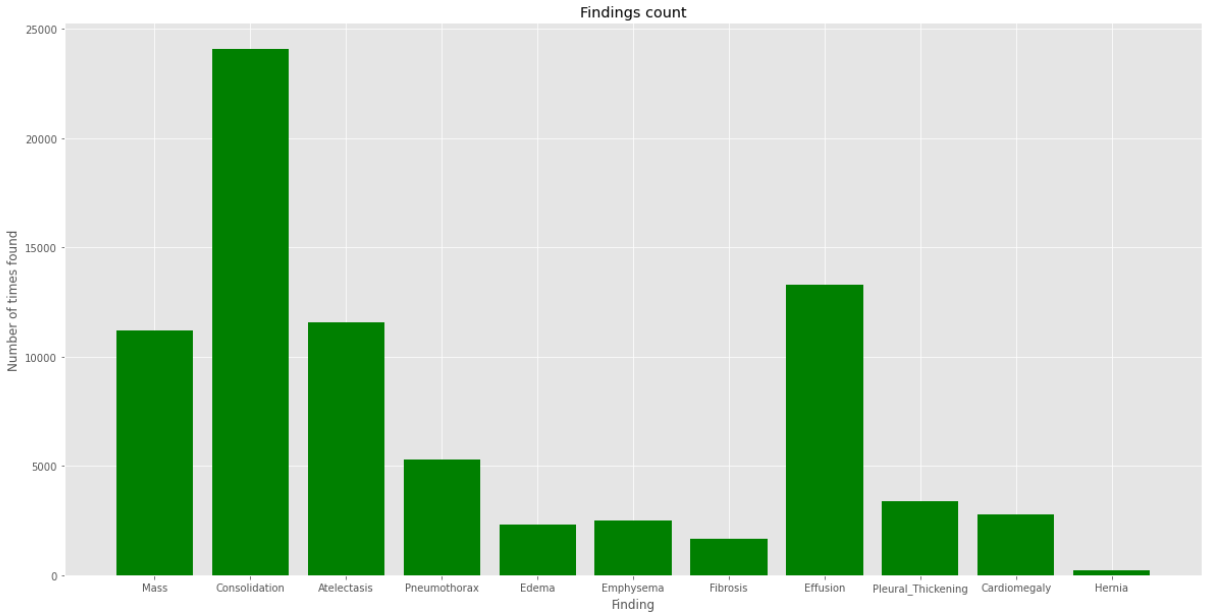
```
Mass 0.5287181735038757
Consolidation 0.5373497009277344
Atelectasis 0.6655040383338928
Pneumothorax 0.5640718340873718
Edema 0.3496096134185791
Emphysema 0.5290770530700684
Fibrosis 0.458153635263443
Effusion 0.33830177783966064
Pleural_Thickening 0.458049476146698
Cardiomegaly 0.07096665352582932
Hernia 0.4884214997291565
```

The outputs of each layer were displayed, as well as the heatmaps of activations for all filters overlaid on the input image for the layer.

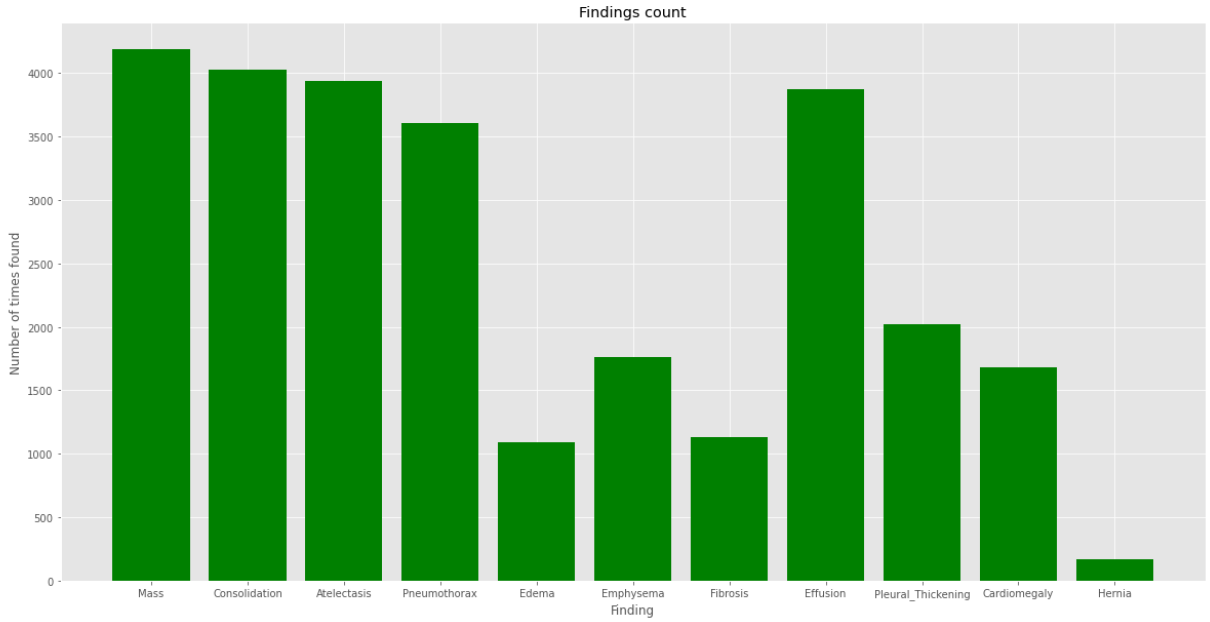


The dataset contained 112,120 images and a CSV table containing the file name and its corresponding finding labels. The images from the dataset are all of the same 1024x1024 dimensions. To better process the information, 11 columns were added to the table, each representing a possible finding label. An image that was positive for some finding would have that corresponding column value set to 1 and 0 if that finding was not present. There was a significant data imbalance found in the dataset, with around 60,000 images containing no findings. There were also some findings that were much more common than others.

Since the dataset was so large, it was decided to undersample the top four most common labels. Any images containing no findings were removed, as well as a large percentage of images with labels containing the findings: Mass, Consolidation, Atelectasis, Effusion. Around 20,000 images remained after undersampling.



Distribution of labels before undersampling



Distribution of labels after undersampling

The dataset was divided into a training set, a validation set, and a test set using a 60/20/20 split, respectively. A data generator was used to downscale the images and split the sets into batches in

order to avoid unreasonably long training times and high memory usage. The training set batch size was set to 64.

The first model used an architecture inspired by AlexNet.[2] Its data generator downscaled the images to 256x256 pixels and set them to grayscale. This color channel was chosen to avoid taking in any unnecessary data, considering the X-rays are already in grayscale. The architecture consisted of five convolutional layers connecting to three dense layers. After every convolutional layer, batch normalization was applied to help speed up learning. The first, second, and fifth layers also had a max-pooling layer to determine the most important features of the previous feature map. The fully connected first fully connected layer was a dense layer containing 4096 units, followed by a dropout layer with dropout rate 0.5. Another dense layer was added after containing 1000 units, followed by an identical dropout layer. The output layer was a dense layer containing 11 units for the 11 labels in the data with a sigmoid activation function to return probability values.

The second model uses the built in architecture MobileNet. It only accepted an input with the shape (224, 224, 3), so a separate data generator was used to downscaled the images, and all three color channels were preserved. Its output layer is a dense layer containing 1000 units, so an additional dense layer was added with 11 to fit the number of labels in the study.

Initially, a binary cross entropy function was used as the loss function. A weighted loss function was used later in an attempt to reduce the rate of false negatives.[3]

[2] Architecture from https://engmrk.com/alexnet-implementation-using-keras/
[3] Fuction found at https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/binary-crossentropy

Experiment and Results

At first, the original dataset without undersampling was used. A model was created using the AlexNet architecture, the Adam optimization algorithm with a learning rate of 0.01 and a binary cross entropy loss function. It was trained for 20 steps per epoch for 10 epochs to get a gauge on how well the architecture applied to the task. I noticed that the binary accuracy was strangely very high even in the early epochs. The model was evaluated after training and had a binary accuracy of 0.9 and an absolute accuracy of around 0.3. After looking at individual predictions for a few examples, I realized that the model was almost always returning values extremely close to 0 for every finding. This was because without undersampling the dataset, around half of the images contained no findings. It is also uncommon for an image to have more than one or two findings, so it is much more likely for a finding to come up as negative for an image. I believe that this resulted in the sigmoid curve being heavily biased to classify a finding as negative. In addition to the problem of an unbalanced dataset, a binary cross entropy function was used as the loss function. The log loss of the sigmoid curve representing probabilities results in a huge penalty when a prediction is a false positive, resulting in the model preferring to output very low probabilities.

```
Actual:      [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Predicted:   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

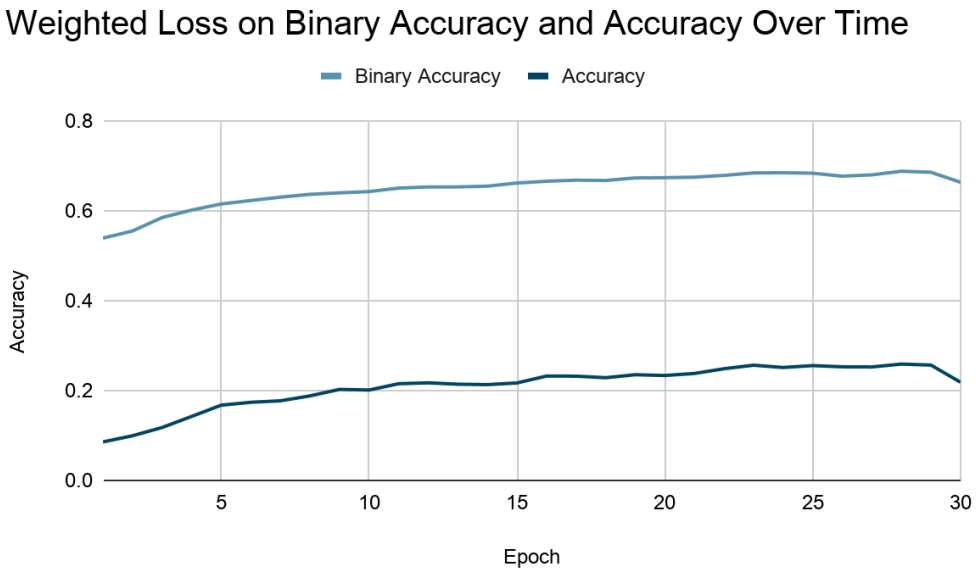
The binary accuracy of this prediction is 0.90
The absolute accuracy of this prediction is 0.00

I first tried to undersample the original dataset to get an improved distribution of positive and negative findings, but it showed minimal improvement, since the individual findings distribution was still heavily skewed towards being negative. The model had a high binary accuracy but also

a high rate of false negatives. I decided that it would be better to decrease the rate of false negatives at the cost of increasing the rate of false positives.

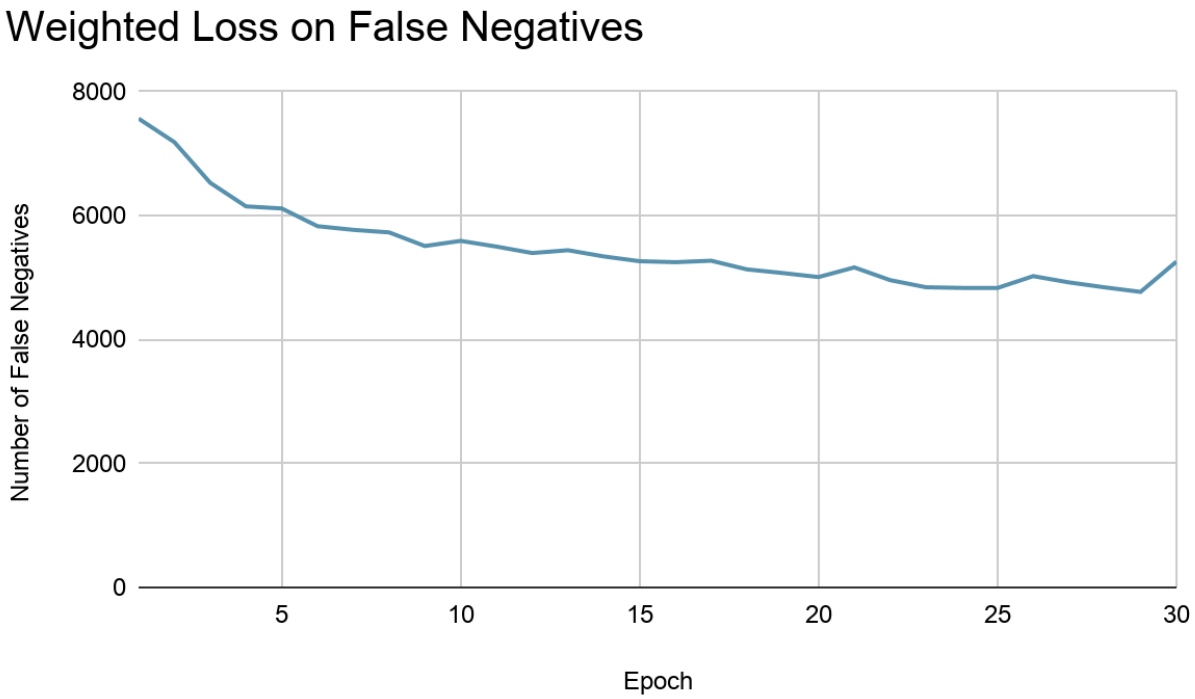
A new loss function was used to assign a higher loss for false negatives. It was the same as the binary cross entropy formula, but weighted the positive and negative losses. This was done by calculating the frequency of each label being positive or negative, and multiplying the log losses by the frequencies for each label. The positive loss and negative loss were then added together to create the total loss.[4]

Using this weighted loss function, a new model was created using the AlexNet architecture and Adam optimizer with a 0.01 learning rate. It was trained for 200 steps per epoch for 30 epochs. A binary accuracy of around 0.68 and absolute accuracy of 0.25 was observed. The lower binary accuracy can be attributed to increasing the likelihood of having false positives. The absolute accuracy was around 0.05 better than the previous model that used the unweighted binary cross entropy loss function.

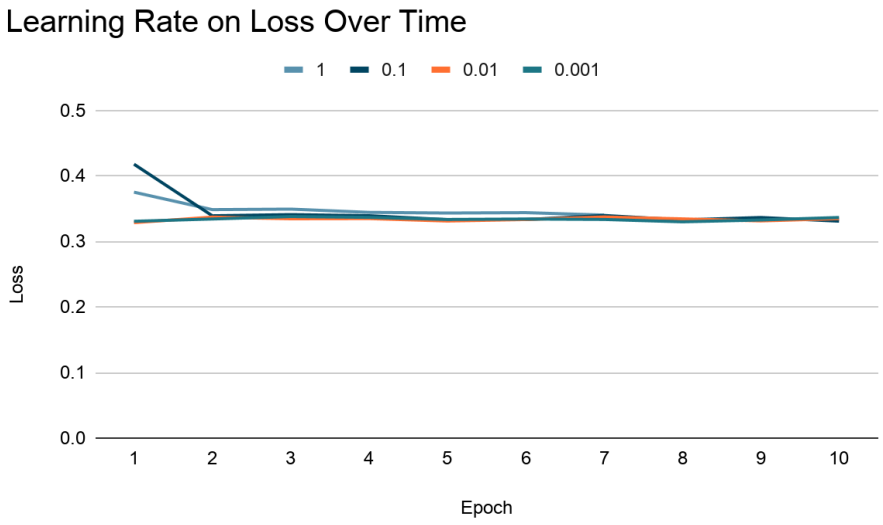


[4] https://towardsdatascience.com/understanding-binary-cross-entropy-log-loss-a-visual-explanation-a3ac6025181a

It was evident that the rate of false negatives also declined significantly.



I also changed the learning rate of the Adam optimizer to determine value to use, but I was strangely unable to find a very large difference between the four I tested. This could be due to the large number of labels I am training for and the relatively low number of epochs each learning rate was run for.



The performance of this network did not achieve “good” performance. While the model using the unweighted loss function technically performed with a very high binary accuracy, it is important to note that this metric is not representative of its actual usefulness, since it happily assigned every image to be completely “healthy”. When using the weighted loss function, results were promising after initial training. Both accuracies were increasing and the rate of false negatives was decreasing. Without hardware limitations, it would be interesting to train the model for 100 or 200 epochs and observe its performance then.

Conclusions

It was difficult to account for the imbalance of labels in the dataset. A balanced dataset could certainly result in much better performance. This could be done by oversampling the underrepresented labels or by reducing the overall number of findings to study. It took me a few days to determine what loss function to use and learn how to correctly calculate the weighted loss when creating the new function. The size of the dataset was a source of many problems, many of which relate to memory limits of online notebook environments. It was also difficult to test for the effectiveness of a certain network architecture, since training time took so long. With a batch size of 64, a single epoch with 200 steps took around five minutes to complete. The 30 epochs it was trained for finished after around three hours. I would like to train that model for much longer than 30 epochs, since it appeared to produce promising results.

Appendix: Code Instructions

The dataset I used was on Kaggle, so I used their online environment for this project. I was unable to run the code on Google Colab due to memory limits when unzipping the ~40GB dataset. The first section labeled “Downloading Dataset” provides the code to download the

Kaggle dataset to your personal Google Drive. I was able to successfully download the dataset to Google Drive as a zip file, but encountered memory issues in Colab when unzipping. The code in that section should work if the user has higher memory limits.

Instructions to run in Colab (unstable)

1. Create a Kaggle account
2. Go to Profile -> Account -> Create New API Token
3. Open the notebook in Colab
4. Run the first code block to mount Drive
5. Run the second block and upload the kaggle.json file
6. Run the third and fourth blocks to download the dataset to the specified Drive location
7. Run the fifth block to unzip the file

Instructions to run in Kaggle (recommended)

1. Open the notebook in Kaggle
2. At the top right corner, click “Add data” and a menu should open
3. Click “Search by URL” at the top right corner of the menu
4. Paste the following URL: https://www.kaggle.com/nih-chest-xrays/data
5. Click “Add” next to the dataset
6. Only run code that appears after the “Processing Data” header

To run the test comparing learning rates, run all of the blocks under the following headers: Processing Data, Cleaning Data, Undersampling, Creating Sets, Generators, MobileNet. The code under the MobileNet header will have to compile and fit new models- each learning rate test is expected to finish before 15 minutes.