Detecting Pneumonia within CT Scans Using Convolutional Neural Networks

Graham Chickering

November 19, 2020

Abstract

Convolutional Networks are a specific type of Neural Networks that have shown to be particularly effective at being able to identify distinct objects within images. This technique can be used to identify different sorts of condition, such as pneumonia, within medical images as well as detect other sorts of tumors or cancers. In theory this type of network is designed based on how the human brain works and the idea that multiple levels of neurons are connected together in order to detect and identify images. In practice though, running and training these types of neural networks can be very computationally expensive and require large amounts of memory and processing capabilities if working with a very large dataset. Especially when working in R which has limited memory capabilities, trying to run and train this type of model can be very slow and ineffective. Thanks to developments in cloud computing such as Google Cloud Storage and Tensorflow being able to store and run these types of models within R can become much faster and more efficient when trying to analyze large amounts of data. While there is more work to be done, this project shows how to create an infrastructure that efficiently stores data and then train convolutional neural networks when working with the R Studio Environment.

Keywords: Convolutional Neural Networks, Google Cloud Storage, Tensorflow, Keras

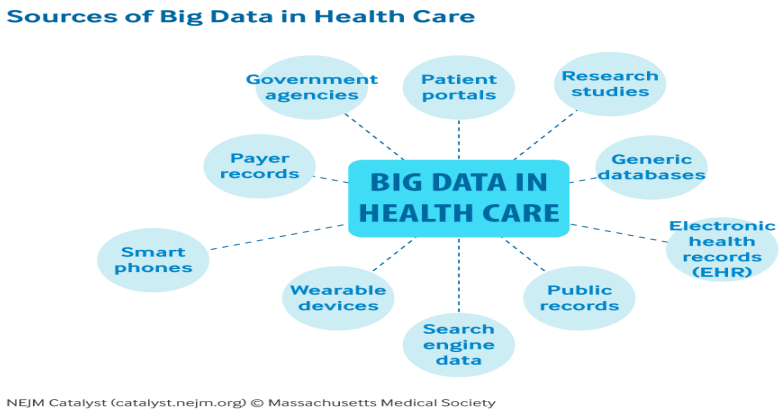


Figure 1: Big Data in Healthcare

1 Introduction

It has been estimated that roughly 80% of health care data is unstructured data, which can come in the form of videos, sensor data, images, or text. Although hospitals and researchers used to have a hard time extracting insights from this type of data, with the recent advances that have been made in data science and handling big data, this has created new application areas within the health care industry in sectors such as genomics, drug trials, predicting patient health, and medical imagery (See Figure 1 NEJM (n.d.)). Medical imaging research in particular has made significant progress recently with researchers being able to use different machine learning algorithms to detect different types of lesions and cancers from CT and other types of scans. In particular the advancements of convolutional neural networks to identify whether or not someone has pneumonia has become extremely promising and can be used to potentially help doctors identify whether or not someone has pneumonia that they might have missed, and reduce the amount of hours and amount of expertise required to view CT scans.

When working with medical imagery data sets one of the first problems someone may run into is how to process and handle these large data sets. When trying to perform analysis on small and medium sized data sets within R, one rarely run into complications that would be attributed to how R is loading and dealing with the data itself. But what happens when one moves from the world of medium sized data to the world of Big Data and large data sets? While most of the time one can load data into the R Studio environment without

issues and without having to worry about whether the entirety of our data can even be loaded in, one may begin to run into complications the larger the data set becomes. If one ends up crossing into the threshold where R can no longer store all the data in an effective way, there are multiple potential solutions in the forms of choosing random subsets of the data, buying a computer with larger memory, or use parallelization and using multiple clusters to perform the analysis. It is this solution of choosing random subsets of data, using the Keras and Tensorflow R packages, that will allow me to work with and convert large files of image data into a form that models can be trained on them.

On top of the issue of trying to run analysis on large data sets in R itself, is the issue of how to best store and load the original information and data. Often data sets are small enough that they can be stored on your local computer in a folder that is then uploaded into the R Studio Environment itself, but what should one do as the size of the data set substantially increases and one no longer wants to store large data sets directly on their machine. One solution to this problem is to take advantage of a cloud computing service and store the data directly in the cloud, freeing up space and memory on your personal computer. By storing the data on a cloud computing service, this becomes especially useful when a project begins to get scaled up whether that is through adding new members to work on the project or when more and more data gets added to the project. In this project I will take advantage of Google Cloud Storage to store my data, saving space on my local machine.

By combining Google Cloud Storage with Keras and Tensorflow, this will allow me to utilize a large data medcical imagery data set that is made up of CT scans. This data set will then be used to train and create convolutional neural networks that will try to identify whether or not someone has pneumonia from the images.

This project will allow me to answer the questions of what is the best way to store large data sets and perform computationally expensive analysis on those data sets? How does one handle and process images so that analysis can be performed on them, and how effective are convolutional neural networks at identifying pneumonia within CT scans?

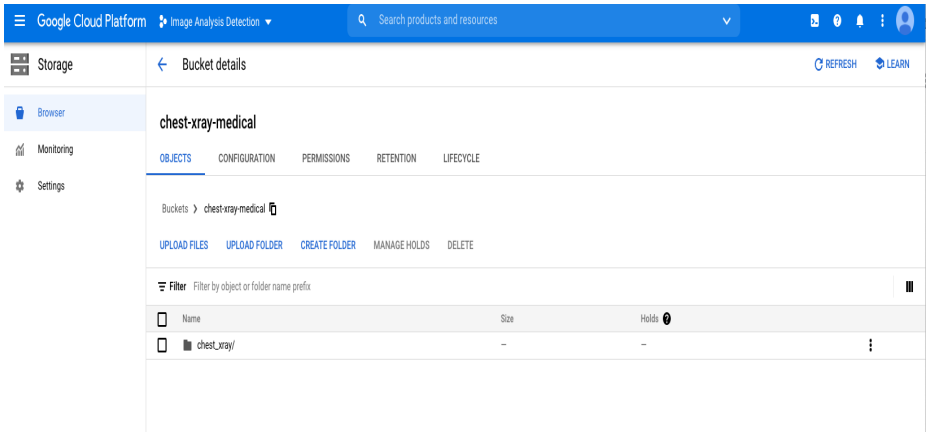


Figure 2: Google Cloud Storage Platform

2 Google Cloud Storage

When trying to work with Big Data, one of the first questions one has to answer is what is the best way to store and access this data. While smaller files can be stored directly on your computer and eventually loaded into R Studio to perform analysis on, when data moves into the gigabyte, terabyte, or even petabyte range one may not not want to store this data directly on their machine and use of large chunks of their limited memory that is available. With the advancement of cloud service solutions in recent years though, one can now use a platform such as Google Cloud Platform, Amazon Web Services, or Microsoft Azure, to store large amounts of data directly on these platforms and take advantage of these companies large data warehouses for a small cost. This can allow one to free up space and memory on their own personal machine and access the data directly from these servers whenever they desire.

For this project, I am going to focus on how to setup and store medical imagery in Google Cloud Storage and learn how this can be connected to R Studio so that I can then perform analysis on these images. The data set I will be working with is a labeled Chest X-Ray images (Daniel KermanyDaniel (2018)) which will be used to detect and classify whether or not someone has pneumonia. This data set is roughly 2 GB in size and contains CT scans of patients who either have or dont have pneumonia. Due to Github having maximum storage limit of 1 GB, I wanted to look into ways that would allow me to work with a data set of this size and not have to upload the data directly into Github itself. One

of the solutions for this was to use Google Cloud Storage and take advantage of the free credits that Google offers for new users using their service. By uploading and storing the data set directly onto the Google platform, this meant I could delete the data set off my computer for the time being and free up memory space (See Figure 2 for an example of the Google Cloud Platform Console).

In order to actually work with this data in R Studio though, a connection between R and Google Cloud was required to be setup. By utilizing the *googleCloudStorageR* package, I was able to setup a connection that allowed me to download the data from my Cloud Storage bucket and onto my desktop where it could then be read into R Studio without having to store the files themselves within R Studio (See Apendix for code on how to setup this connection). This allowed me to not have to store the data in my actual Github repository but still be able to perform analysis and build models on the image dataset.

3 Image Transformation and Tensorflow

Once the images are in a place where they could be loaded into R, one needs to put the images into a format where R can actually perform analysis on them. For images, this means making it so they all have the same underlying which can require different image transformations and adjustments such as cropping, brightness, contrasting, changing the color scale, or resizing an image. This sort of data augmentation, when performed on all the images in the data set can help create a more consistent form between all the images. For the images in my medical imagery data set,this primarily consisted of resizing the images on a pixel by pixel basis so they were all the same size, and then converted them to a grayscale color. This made the underlying structure of the images the same for all the images in the data set. See the Figure 3 for an example between a patient with and without pneumonia after adjustments were made.

After getting all the images into the same structure, I needed to convert them to a form where I could actually perform analysis on the images. In order to do this I relied on the *tensorflow* R packages. Tensorflow is a "scalable and multiplatform programming interface for implementing and running machine learning algorithms" Raschka & Mirjalili (2015). Tensorflow allows execution on both CPU's and GPU's which help speed up processing time

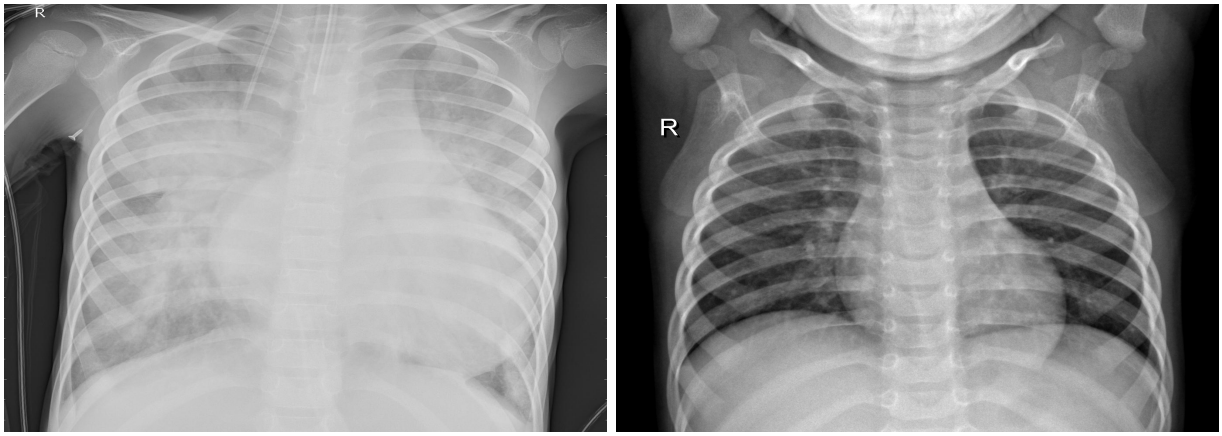


Figure 3: Patient CT Scans. At left, patient with pneumonia. At right, a patient without pnuemonia

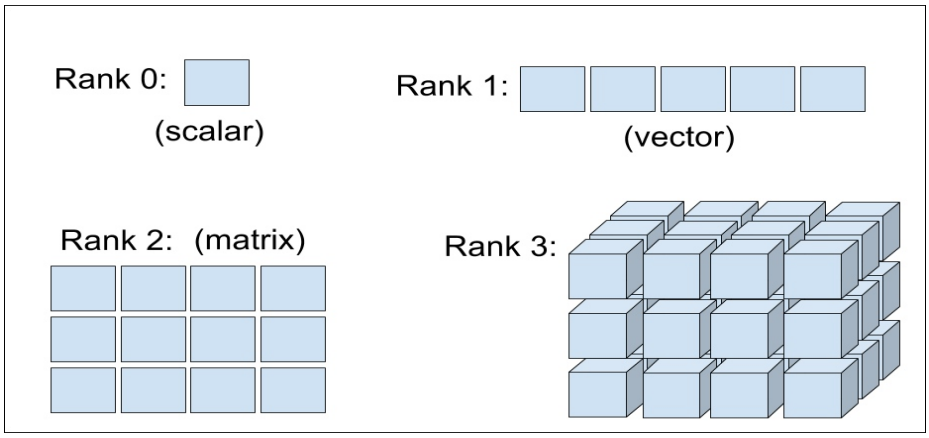


Figure 4: Tensors in Tensorflow

on complicated algorithms. This package is built around a "computation graph composed as a set of nodes where each node represents an operation that may have zero or more input or output. A tensor is created as a symbolic handle to refer to the input and output of these operations" Raschka & Mirjalili (2015). A tensor is best understood as either a scalar, vector , matrices and so on, which all correspond to a different rank tensor (See Figure 4). These tensors are created from the values in the data you are working with and then are used to build and create the complex models one wants to work with.

For the images in my data set, by using Tensorflow I was able to convert the images into a set of tensors that represented the pixels of the images themselves. Since the images had been resized to a 64x64 pixel picture in grayscale coloring, this became a 64x64x1 tensor for

each image. These tensors were then combined with their appropriate label for the type of image they were, either Normal or Pneumonia, creating a nicely formatted data set where we could then start to create models to best classify the data.

4 Keras and Convolutional Neural Networks

As we could see from the prior pictures of someone who had pneumonia versus someone who did not have pneumonia, it can be very hard to discern whether or not someone has pneumonia for a person without the technical training and expertise to identify pictures of pneumonia from looking at the CT scans. This ultimately requires doctors and those with the expertise to spend more time looking at the scans, rather than spending their already limited time directly helping patients. One way to help doctors to get back to directly helping patients is by utilizing machine learning to identify and detect different diseases or ailments within CT scans. By training and creating models that are able to discern between a normal or healthy scan versus someone who has a lesion or has pneumonia, we can begin to use artificial intelligence to alleviate some of the extraneous work that doctors are required to do.

4.1 Keras

For this project, I was specifically focused on creating a model that would would be able to discern between someone who has pneumonia versus someone who does not have pneumonia. In order to do this I utilized the *keras* package in order to build a convolutional neural network to be able to distinguish between the two different diagnosis. Keras is a "high-level neural network API that is built to run off other libraries such as Tensorflow to provide a user-friendly interface to building complex models" Raschka & Mirjalili (2015). Keras, when working with Tensorflow, helps to provide the framework to begin building complex neural network models. This package will allow me to be able to not only train and build the model, but be able to test the model on another subset of images and be able to begin to predict whether or not someone has pneumonia from looking at a specific image. Specifically I used the keras package to build out a convolutional neural network

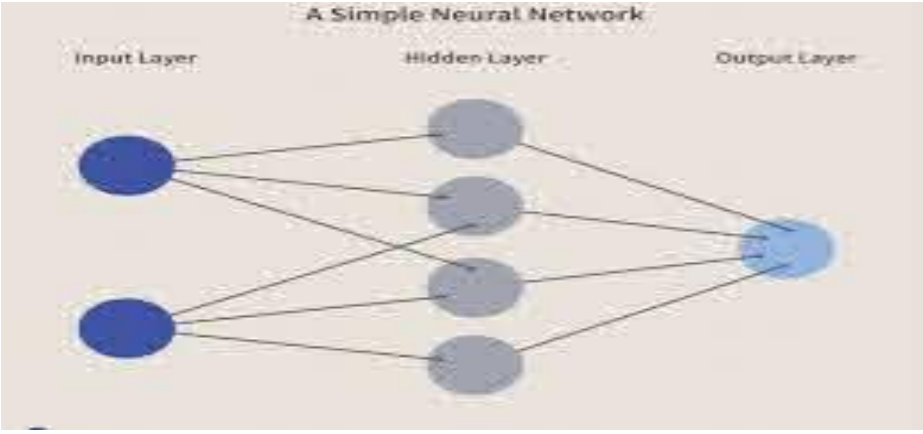


Figure 5: Neural Network example

that have been shown to work extremely well for image classification tasks.

4.2 Neural Networks

Convolutional Neural Networks (CNN) at a high level are a form of deep learning thattakes an image as an input and eventually classifies it under a certain category. These types of networks are used heavily to perform tasks such as facial recognition, object detection, and image classification to just list a few. CNN's are a specific type of neural network, which falls under the branch of machine learning called deep learning. Neural networks are extremely popular today due to recent advances in both the algorithms for the models and the computer architecture which allows for much faster processing times of these complex models.

The figure above shows an example of what the underlying structure of a neural network looks like (See Figure 4 Chen (n.d.)). The neural network show in this example consists of one input layer, one hidden layer with 4 hidden units, and then one output layer. The input layer would be the exact data you are feeding into the model, so for this picture this would mean there was two examples being fed into network to train the model. Then we can see how each of the input layers are connected to three of the gray nodes in the hidden layer. While this model only has one hidden layer and the inputs are only connected to 3 of the hidden units, in other models one could create multiple hidden layers, each with a different number of hidden units, and the number of units that are connected together from layer to

layer can vary as well. After that we can see that there is one output layer for this example, but other networks can contain varying numbers of output units as well. Neural networks in general can contain differing number of hidden layers, hidden units, input and output units and vary based on the data they have and the problem they are trying to solve. In general data is fed into the model, it is passed through the network using weights, and then often trained using a technique called backwards propagation. This technique is able to optimize the model by comparing the result found by the model to the true results, and then working backwards through the model to update the weights so that the model can better classify the input data. Neural networks in general have been shown to be extremely successful at a broad range of tasks such as natural language processing to self-driving cars and everything inbetween.

4.3 Convolutional Neural Networks (CNN)

As mentioned earlier, CNN's a specific form of Neural Network that is based heavily on how the visual cortext of the human brain works when recogizing images, which is what allows it to perform extremely well on image classification tasks. At a high level CNN's work by "combining the low level features in a layer-wise fashion to form high-level features" Raschka & Mirjalili (2015). So rather than just looking at each pixel of an image separately, this model works to combine pixels into distinct features that can then be used to identify exact objects within the images. In order for the model to actual identify distinct features it relies on an idea called feature mapping that groups patches of pixels together in the image and combines them into one feature in the new feature map. This is based on the underlying assumption that in the context of image data, "nearby pixels are typically more relevant to each other than pixels that are far away from each other" Raschka & Mirjalili (2015).

In order to actually perform this feature mapping though a CNN relies on creating a series of different types of layers in the form of convolution layers, subsampling layers, pooling layers, and dropout layers.

4.3.1 Convolution layers

Convolution layers is one of the first layers that begins extracting features from the input images. This layer works by taking an input matrix that represents the image and a filter matrix, of potentially a different size, with a set of weights. These two matrices are multiplied together to create a feature map that begins to identify the low level features such as edges, blurriness, or sharpness of the image.

4.3.2 Subsampling and Pooling Layers

Another type of key layer is the subsampling and pooling layers. Usually the feature map that is created from the previous convolution layer is then fed into a subsampling/pooling layer. These layers work by combining small subsections of the feature map in order to simplify the feature map. The advantages of this include leading to higher computational efficiency by decreasing the size of the features and the number of parameters that are required to learn, as well as introducing local invariance that helps to generate feature that are more robust to noise from the input images MathWorks (2019).

4.3.3 Dropout layers

Dropout layers are then used to prevent over-fitting of the data set. It can be very easy to create a CNN that gets over-trained but then fails to perform well on the testing set of data. To prevent over-fitting and make the model work well for a broader range of images for general performance, dropout layers are introduced as a form of regularization. Dropout is usually applied to hidden units of layers and works by "during the training phase of a neural network, a fraction of the hidden units are randomly dropped at every iteration. This dropping out of random units requires the remaining units to rescale to account for the missing units which forces the network to learn a redundant representation of the data" Raschka & Mirjalili (2015). This makes it so the model is more generalizable and robust to changes in patterns in the data and prevent over-fitting.

4.3.4 Challenges

When working with CNN's, these different types of layers that the network can be built off can all be included multiple times and in different orders. For example one could create a network of just a convolution layer and a dropout layer or someone could create a model that is convolution layer, pooling layer, convolution layer, and then two dropout layers. There is no set rules around what order or how many different layers your network can have. On top of having unlimited variations of the type of layers within the network, there is also unlimited parameters that one can choose in terms of the size of the feature map, the number of units to pool together in the pooling layer, or the number of units to dropout in the dropout layer to just list a few examples. Due to having so many input parameters that can be changed and altered when it comes to building the model, finding the best possible input parameters and underlying structure of the model can become computationally expensive very quickly.

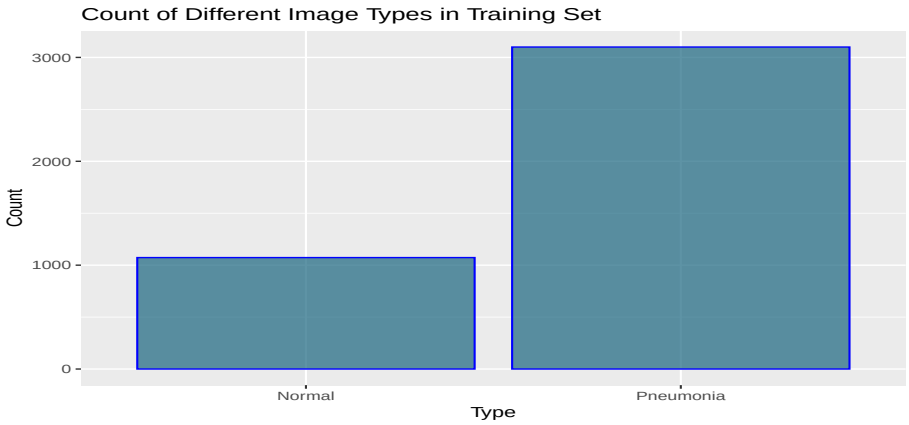
On top of CNN's being very computationally expensive to run, being able to interpret the outputs and inputs of individual layers of the network is its own separate issue. Often these types of models are referred to as "black box" models because while a human can understand the input and outputs of the models, it is often hard to decipher and understand what exactly the model is doing inside the box. While there has been promising developments in this area such as Visualizing Activation Layers and Occlusion Sensitivity, this still remains an area of the field that a lot of future work can be done in Sarkar (2019). While CNN's have been shown to do an incredible job at image classification tasks, optimizing and understanding these types of models is an area that can cause problems and issues.

4.4 CNN's with Medical Images

For my project, I wanted to use Convolutional Neural Networks to see how well this type of network could perform on the task of trying to classify whether or not someone had pneumonia based just of a CT scan. At this point I have already gotten my images into a form that allows them to be passed as inputs to build a model, so the next step was for me to begin training the model.

To begin training the model I started off by first splitting my data into training, validation, and testing sets. The training set would be used to train the model, the validation set would be used to tune the parameters in the network, and then the test set would then be used to assess the performance of the final model and see how generalizable the results are when given separate input images.

After splitting up the data into their different sets I found that overall the data set contained many more pictures of people than pneumonia than people who did not have pneumonia. In fact the training data contains 74% picture of people with pneumonia and 26% being normal images.



What this tells us is that if a random person were to guess that every picture was a picture of pneumonia, that they would be correct 74% of the time. So when I build out my models, I will be looking for a model that is able to achieve better than 74% accuracy to say it is an improvement over a baseline random guessing model.

4.5 Model 1

Now that I had an understanding of what my training set looked like, I wanted to try building my first convolutional neural network. For my first model I decided to build a model that had one convolution layer, one pooling layer, one flattening layer, a dropout layer that removed half of the available units available in the previous layer during each new training iteration, and then finally one dense layer. The flattening layer works to replace all dimensions of the previous tensors down to one dimension, which is the dimension size we want our output layer to be. The dropout layer is used to make the model more

generalizable and requires the model to fit the units with only half the units of the previous layer available during any iteration. The final dense layer is used to create a layer of units, in this case one unit, where every unit in this new layer is connected to every unit in the previous layer, making it densely connected. See Figure 5 for what this model looks likes. From looking at Figure 5 one can see that there are 13,121 trainable parameters. These are all the weights between the different layers that the model will try to optimize during each training run.

```
model1<- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3,3), activation = "relu",
                input_shape = c(64,64,1)) %>%
  layer_max_pooling_2d(pool_size = c(3,3)) %>%
  layer_flatten() %>%
  layer_dropout(rate=0.5) %>%
  layer_dense(1, activation="softmax")
```

```
model1 %>% compile(
  optimizer = "adam",
  loss = "binary_crossentropy",
  metrics = "accuracy"
)
summary(model1)
```

```
## Model: "sequential"
## _________________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =========================================================================
## conv2d (Conv2D)              (None, 62, 62, 32)        320
## _________________________________________________________________________
## max_pooling2d (MaxPooling2D) (None, 20, 20, 32)        0
## _________________________________________________________________________
```

```
## flatten (Flatten)                    (None, 12800)                    0
## _________________________________________________________________________
## dropout (Dropout)                    (None, 12800)                    0
## _________________________________________________________________________
## dense (Dense)                        (None, 1)                        12801
## =========================================================================
## Total params: 13,121
## Trainable params: 13,121
## Non-trainable params: 0
## _________________________________________________________________________
```

After training this model we can see that although it starts at around being 75% accurate on both the training and validation sets, that even after 5 epochs that the model does not do any better at being able to discern between whether or not an image is normal or pneumonia than just random guessing. This suggests that either the model does not have enough layers to it and is not able to extract distinguishable features from the images or that there were not enough epochs to train the model. It is also worth noting that this model took 2 and a half minutes to complete its training, with roughly 30 seconds per epoch. This long run time is due to the large number of trainable parameters in the model.

4.6 Model 2

Since model 1 did not do any better than one would do than just randomly guessing, I decided to try to improve upon the first model by adding a second convolution layer after the first pooling layer, as well as a second dense layer at the end of the model. I also decided to increase the number of epochs from 5 to 10 to see if allowing the model a longer period of time to train itself would help improve the accuracy of the model at all. My hopes were that adding these new layers would help the model discover features in the images that it wasnt able to detect with the first model and adding more epochs would give the model more time to find these features. Figure 7 shows how the new model is constructed. We can see that by adding these two new layers to the model that there are now 34,659 parameters that the model is going to try to maximize during its training process.

Even after adding a second convolution layer and a second dense layer at the end, we can see that this model again does not do any better than the previous model. One can see that the accuracy stays right around 75% for both the training and validation sets, and that even after increasing the complexity of the model slightly and increasing the number of epochs that the model is still not any better than guessing Pneumonia every time. This model also took roughly 5 minutes to run with every epoch taking 30 seconds to run.

4.7 Model 3

Since my second model still did not do much better than someone randomly guessing, for my third model I decided to increase the complexity of the model substantially. Since the previous models did not appear to be able to detect any distinguishable features from the images, by increasing the number of layers in the model I believe it should help the model finally be able to detect the underlying features. For this model I decided to have 4 pairs of convolution, pooling, and dropout layers, followed by a flattening layer and then two dense layers. As we can see from Figure 9, this model now has 98,017 trainable parameters that the model will try to optimize.

As we can see from this output, the model is finally able to discern between images that contain pneumonia and normal images. As we can, over the course of 10 epochs the model is able to increase it accuracy from under 75% to close to 90% by the 10th epoch. From the graph we can also see that the validation accuracy score is always slightly under the training accuracy score, which suggests that the model might be slightly over-fitting the data, but because I already included 4 dropout layer in the model and the scores are still relatively close, this is not something to worry about too much. Now that I have found a model that is able to achieve close to 90% accuracy, I am going to move on and try this model on the testing set. This model also took close to 5 minutes to run even though the number of trainable parameters increased substantially.

4.8 Results

After running model 3 on the testing set, the model was able to achieve an accuracy score of 88.94% and a loss score of 0.325 (which would ideally be at 0 if there was perfect

accuracy). The testing set contained 234 normal images and 390 pneumonia images so if one were to guess pneumonia every time they would be correct 62.5% of the time. While in an ideal world the model would be able to achieve a perfect 100% accuracy and 0 loss, being able to train a model that achieves an accuracy of 89% on a task as challenging as being able to identify whether or not someone has pneumonia is a very promising result.

When we break down the accuracy score further and look into how the model did at identifying the two different classes one can begin to look into areas where the model does very well and where the model struggles. By looking at the confusion matrix in Figure 11, we can see that the model does an extremely good job of identifying when the image is of someone who has pneumonia, being able to predict is correctly 96.4% of the time. We can also see that the model is only able to correctly predict whether someone is normal at a 76.5% of the time. What this output tells us is that the model is more likely to identify a patient as having pneumonia when they have do not have it, rather than the opposite scenario of telling a patient they do not have pneumonia when they in fact have it which is probably the more dangerous scenario. When thinking about why the model does such a better job at identifying patients who have pneumonia, it comes back to the type of data we used to train the model. Since the training and validation data contained 3x as many pictures of pneumonia, it makes sense that the model does a much better job of being able to recognize when someone has pneumonia. So while being able to achieve 89% accuracy with a convolutional neural network is very promising, these results show that there is future work that can be done that would be able to improve the model's accuracy even more.

5 Conclusion and Future Work

- working with google cloud platform environment, not having to store the data on my computer at all -model could have been optimized
- further application areas, such as lesions and other things that can be detected in ct scans (enhancing images themself)`https://healthitanalytics.com/news/deep-learning-model-can-enhance-standard-ct-scan-technology`

‘

References

Chen, J. (n.d.), ‘Neural networks’.

URL: *https://www.investopedia.com/terms/n/neuralnetwork.asp*

Daniel KermanyDaniel, Kang Zhang, a. M. G. (2018), ‘Labeled optical coherence tomography (oct) and chest x-ray images for classification’.

URL: *https://data.mendeley.com/datasets/rscbjbr9sj/2*

MathWorks (2019), ‘Convolutional neural network’.

URL: *https://www.mathworks.com/discovery/convolutional-neural-network-matlab.html*

NEJM (n.d.), ‘Healthcare big data and the promise of value-based care’.

URL: *https://catalyst.nejm.org/doi/full/10.1056/CAT.18.0290*

Raschka, S. & Mirjalili, V. (2015).

Sarkar, D. (2019), ‘Interpreting deep learning models for computer vision’.

URL: *https://medium.com/google-developer-experts/interpreting-deep-learning-models-for-computer-vision-f95683e23c1d*