Problem formulation

The problem involves prediction the presence of absence of COVID-19 using varying X-ray images. The input of this problem is a series of X-ray images of the chest, which goes into a prediction model and the model predicts based on the images if a person has COVID-19 or not. This is an image classification problem which is well suited for the implementation of neural networks. The main problem regarding image classification involves finding useful features (shapes, edges, regions) which are hard to hand craft, however a neural network is able to select those features automatically. Both fully connected and convolutional neural networks have previously been used for image classification problems.

There are several challenges could rise in regard to using fully connected and convolutional neural networks. A fully connected model requires weights which would problematic for larger sizes images, having such large number of parameters creates problems such as slower training time, or risk of overfitting. The main difference between a fully connected and convolution model is that the main image matrix is reduced to a matrix of lower dimension in the first layer through an operation called convolution. The problem for this assignment is to investigate the process involved for fully connected vs convolutional neural networks and improve AUC by various optimization techniques and tine tuning design.

Fully connected models

- EPOCHS: First I increased the number of epochs from 30 to 40 and then 50, I noticed that the fully connected layer performs better with higher number of epochs, however I observed a plateau in performance around 45-50 epochs so to avoid the risk of overfitting the data I will not increase this parameter more.
- DROP OUT: Since I was afraid that having many epochs would still overfit my data, I then added a drop out (which is a regulation technique to reduce overfitting by resampling the output of the layer under dropout. Adding 2 dropout layers did decrease model performance, which makes sense if we assume that the model was overfitting before. I then removed one dropout (I had 2 before) and saw better results. The accuracy was gradually increasing while loss was gradually decreased, which was not the case when I had two dropouts.
- LAYERS: I decided to add ONE dense layer (dense (16)) first and observed that my model performance decrease a little. Then I tried increasing the dimensionality (dense (128)) of the output space, doing this significantly improved my model performance. Generally adding more layers would add more weights to the model and improve score but here I'm assuming the model might be overfitting if it did not initially perform well with the previously added low dimensional dense layer. Also since I'm going from dense (64) to dense(32) and then dense(128), it means more emphasis is put on important features of dense(32) going to dense(128) aka could be overfitting. To better resemble how image classification neural nets are structure I will switch the order so that it is dense (32) → dense(64) → dense(128), doing this increased model performance while decreasing rate of loss.
- OPTIMIZERS: I then decided to work on optimizing the model. Leaning rate is a hyperparameter that controls how much change is needed for the model in response to the estimated error each time the model weights are updated. A small learning rate might result in a long training process and a large one may result in learning a not so great set of

weights too fast. The first learning rate I tried was the adaptive learning rate gradient descent (adapts the learning rate, one learning rate per model weight) using the Adam optimizer. I set the Adam learning rate to 0.0001 (default is 0.001) to compare, and saw that the model did not have a good performance and then increased it to 0.01 and got slightly better results however the default still seems to be the better option. I decided to use another adaptive learning rate, so I replaced Adam with 'Adagrad' (decays the learning rate for parameters in proportion their update history) which in general did not preform as well as Adam. I also tried the RMSProp (Root mean square propagation) algorithm (decays the denominator to prevent rapid growth which is a limitation of AdaGrad which decays the learning rate very aggressively), RMSProp performed worse than both Adam and AdaGrad.

Convolution models (CNNs)

A convolutional neural network is generally better than just flattening an array of pixels and feeding to an MLP style model for classification. A CNN can capture the spatial and temporal dependencies of an. Image through the application of relevant filters. The role of a CNN is to reduce the images into a form that is easier to process while not losing feature that for critical for a good performance.

- MODEL 1: is the element involved in carrying the convolution operation in the first part of a convolution layer. The kernels in the convolutional layer are the convolutional filters so kernel size refers to the weight * height of the filter. Since there are no set rules on how to pick a number for the filter, as a general guideline I will make the filter number grow with subsequent layers. First layer (with a lower number of filters than second) will catch a few features (edge, shape, etc…) of the image and next layers will obtain more complex features based on simple ones. I also added the activation function 'reLU' to my first conv2D layer. I used 'reLU' because it slows the network to converge very quickly and allows for back propagation. I then implemented a BatchNormalization() layer to automatically standardize the inputs to a layer, I did this because implementing BatchNormalization() will accelerate the training process and perhaps improve performance via a regularization effect. I also added a Max Pooling layer as a down-sampling strategy (reduces size of resultant image while retaining image information). I then added a flattened (explained above) and dense (explained above) layers. Generally, the performance that I got with this model was better than any of the previous non-CNN models.
- MODEL 2: Added an extra denes layer as well as another convolution layer and increased epoch size which did not make a significant improvement in results but I also noticed that my loss value does not go down to change at all and my model is predicting either all 0's and 1's. To overcome this problem, I decided to first start with a very basic CNN and implement layers as I go.
- MODEL 3: I assume that my CNN model might be too complex for the small dataset that we have. I decided to take out some of the layers and details. To verify that the model works correctly I had to first simplify it as much as possible. After searching for some solutions and recommendations online, I decided to reduce my number of layers to only 1 convolutional layer, then a Max Pooling later, I then flattened the data and added a dense

layer. This made a significant difference in the loss rate and also significantly improved model performance.

- PARAMETER TUNING: In general, the activation function ‘sigmoid’ and ‘relu’ works best in the convolution layer with ‘relu’ performing slightly better than ‘sigmoid’, and ‘relu’ also had a faster computation time. ‘sigmoid’ is very common for image classification CNNs however it does suffer from the vanishing gradient problem which is not a concern with ‘relu’. However, I did note than not including an activation function in the convolution layer does improve results a bit, which is I thought was a bit strange but maybe it implies that the problem is a bit simpler than I thought. I decided to change the image shape to 200x200 and saw a dramatic increase in performance. I also tried different optimizers, however noticed that Adam outperformed all, the default learning rate for Adam was also the best option. In terms of epochs, I found the best numbers to be around 33-45 anything outside of those seemed to under or overtrain the data, however whenever I wanted to use more epochs, I lowered the learning rate when trying to avoid overfitting.