Diagnosing COVID-19 in Chest X-Ray Images Using Various Machine Learning Classifiers

Anthony Krivonos (ak4483)
COMS 4771 Machine Learning

With the advent of COVID-19, it has become a major challenge in Machine Learning to be able to classify infected lungs from healthy lungs using merely x-ray images. We present three methods for classifying x-ray images—Support Vector Machines, Multi-Layer Neural Networks, and Residual Convolutional Neural Networks—and discuss their performance on classifying healthy lungs, bacterially-infected lungs, virally-infected lungs, and COVID-19-infected lungs using 1,127 training images. We also show four methods for pre-processing raw x-ray images and the effects of each technique on classification across each classification method. Finally, we conclude that CORONet, our predictably-named custom residual convolutional neural network, exhibits the greatest classification accuracy on a holdout training data set, leading to a Kaggle submission score of 0.87908.

I. INTRODUCTION

Knee-deep in the greatest pandemic of the past century, our society has had little on our minds besides diagnosing, treating, and finding a vaccine for SARS-CoV-2 (hereby "COVID-19").[1] Using modern machine learning techniques, 350 images of normal lungs, 350 images of lungs with bacterial pneumonia, 350 images of lungs with viral pneumonia, and 77 images of lungs infected with COVID-19 were examined and used to train three models to classify images of human lungs. These models were then tested on 484 test images and submitted to the COMS 4771 COVID Challenge on Kaggle.

A. What are Machine Learning and COVID-19?

This is not a Medium article and we assume that the reader is aware.

B. Heuristic Examination of Images

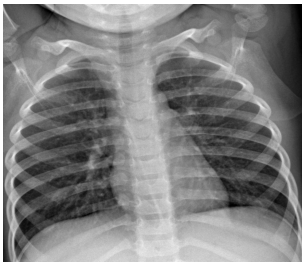
Figure 1 shows 4 images belonging to classes

$$\mathcal{Y} = \{ \text{ normal } , \text{ bacterial } , \text{ viral } , \text{ covid } \}$$

.

The following general observations were made which later guided the pre-processing steps taken before classification. *Disclaimer: These observations are evidently subjective and I am obviously not a licensed medical practitioner.*

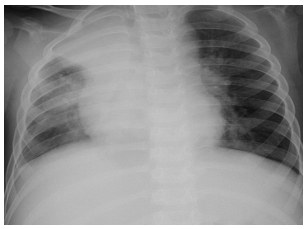
- Normal lung images are usually the most contrastful and least inflammated.
- Viral lungs usually appear deflated and have more strain-related noise around the center.
- Bacterial lungs look very similar to viral lungs, except have more noticeable cloudiness towards the top.



(a) Normal lungs.

(b) Lungs with bacterial infection.

(c) Lungs with viral infection.



(d) Lungs with COVID-19.

FIG. 1: Lung images in training data for each of the four classes.

- COVID-19 lungs have the most prominent deformation towards the bottom.

Regardless of the validity of the above observations, it is crucial to consider that the images in both the training and testing datasets are of different sizes. Thus, resizing algorithms must be considered in the pre-processing stage. Additionally, some level of noise reduction must be implemented in order to reduce/emphasize the "cloudiness" that is present in bacterial, viral, and COVID-19 lungs.

II. PRE-PROCESSING

After developing an intuition for the composition of the images from each class in $\mathcal{Y}$, the first course of action was to determine how to emphasize distinguish-

able features in the X-ray images, like *cloudiness* and *inflammation*. Intuitively, increasing the contrast of all images seemed to be a plausible first step. It was thus decided to use CLAHE (Contrast Limited Adaptive Histogram Equalization) to create a starker contrast in each image.[2]

The next issue pertained to varying image sizes in both the training and testing directories. Four solutions were devised, labeled `Method A`, `Method B`, `Method C`, and `Method D`, and are described as follows. Each of these methods were tried width three different conditions for *threshold* size:

- Resizing to the *maximum* width and height of all training images.
- Resizing to the *minimum* width and height of all training images.
- Resizing to a preset size of 200 pixels, maintaining aspect ratio.

Additionally, images from all methods were post-processed through CLAHE.

A. Method A

First, we found the training image with the *threshold* width and height. Then, we took either $\max\{\text{width}, \text{height}\}$ or $\min\{\text{width}, \text{height}\}$ of the image, and called it the *threshold* size. All training and testing images were subsequently resized to the threshold size, maintaining their aspect ratios. The empty spaces in each image were filled with black pixels.

B. Method B

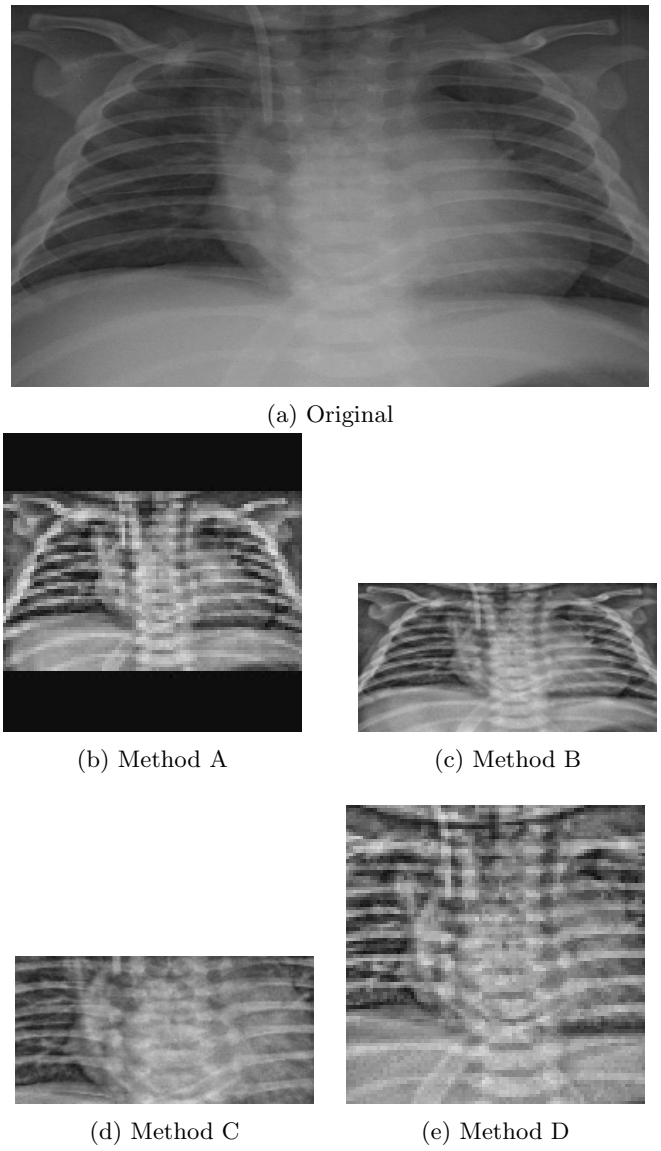
For this method, we applied the *threshold* dimensions to all training and testing images, ignoring aspect ratio.

C. Method C

As in method B, we applied the *threshold* dimensions to all training and testing images, this time keeping each image's aspect ratio.

D. Method D

As in method A, we found the training image with the *threshold* width and height. Then, we took either $\max\{\text{width}, \text{height}\}$ or $\min\{\text{width}, \text{height}\}$ of the image, and called it the *threshold* size. All training and testing images were subsequently resized to the threshold size while covering the square frame and maintaining their aspect ratios.



(a) Original

(b) Method A

(c) Method B

(d) Method C

(e) Method D

FIG. 2: 7th training image (bacterial) run through each of the pre-processing methods

III. CLASSIFICATION

A. Support Vector Machine (SVM) w/ Stochastic Gradient Descent (SGD)

1. *Background*

For my first classifier, I chose the SVM w/ SGD for my familiarity and bias (no pun intended) towards it. Some of the inspiration behind this algorithm was taken from Daniel Leed's Machine Learning course at Fordham University.[3] This algorithm utilized the *one-against-all* technique, in which raw label scores ($-1 \leq s \leq +1$) were found for each of the four classes in $\mathcal{Y} = \{\text{normal}, \text{bacterial}, \text{viral}, \text{covid}\}$ per data point. Then, a final iteration assigned each data point to the class with

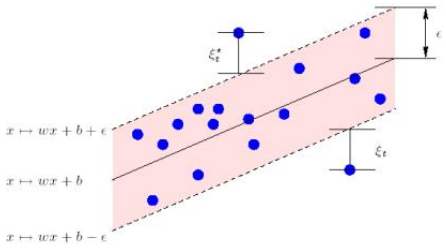


FIG. 3: SVM parameter visualization

the largest raw score. This technique is equivalent to receiving continuous responses on $[-1, +1]$ for each of the following questions and taking the maximum of all responses per data point to find the label:

- "Is it *normal*?",
- "Is it *bacterial*?",
- "Is it *viral*?", and
- "Is it *covid*?".

2. Algorithm

Weights were trained depending on the number of epochs provided. Additionally, a variable C determined the SVM's tolerance for misclassified data points and a variable ϵ defined the tolerance margin in which no penalty was given to errors. These parameters are visualized in Figure 3.[4]

For each of the labels, the following algorithm was run on the entire data set:

- Initialize the weights $w \to 0$ and bias $b \to 0$.
- For a desired number of epochs, for every row x in the training data X:
  1. Flatten the 2D image matrix x into a 1D vector.
  2. If $1 - y \cdot w^T x + b > 0$:

$$w = w - \epsilon \cdot (\frac{w}{\text{epochs}} - C \cdot y \cdot x), \tag{3.1}$$

$$b = b - \epsilon \cdot \frac{w}{\text{epochs}}. \tag{3.2}$$

  3. Otherwise,

$$w = w - \epsilon \frac{w}{\text{epochs}}. \tag{3.3}$$

  4. Set $\epsilon = \frac{\epsilon_{\text{original}}}{\text{epochs}}$.
- Loop through all $\chi \in$ test data. Set the label of $\chi_{i,l}$, where i is the index and l is the "focus" label in one-against-all, to be $w^T \chi$, where w is the learned weights vector.

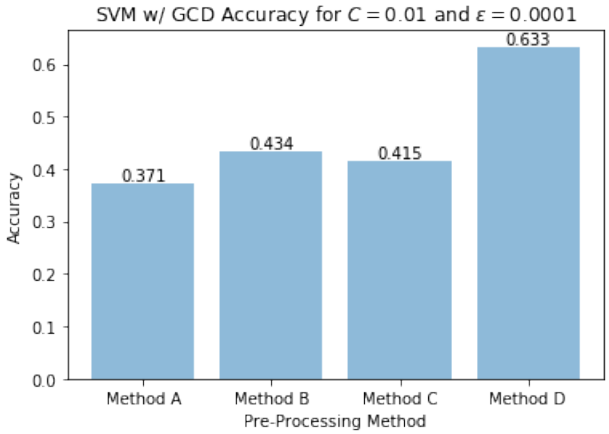


FIG. 4: Results for SVG w/ SGD with k-Fold CV

- Loop through all $\chi \in$ test data once more, this time setting the label of χ_i to $\max\{\chi_{i,\text{ normal}}, \chi_{i,\text{ bacterial}}, \chi_{i,\text{ viral}}, \chi_{i,\text{ covid}}\}$.

3. Results

Figure 4 shows the results of *k-Fold Cross Validation* with $k = 10$ over the training data set for each of the image pre-processing methods.

Evidently, none of these results fared better than the minimum threshold accuracy of 0.80065, and so other methods for classification were examined.

4. Commentary

The most time-consuming part of the SVM approach was tuning parameters C and ϵ, and obtaining the optimal number of training epochs to prevent overfitting. To find accurate parameters, k-Fold Cross Validation was run with $k = 10$ over the training data set pre-processed with Method D for a small number of `epochs`$= 100$ for $C \in \{0.1, 0.01, 0.001\}$ and $\epsilon \in \{0.001, 0.0001, 0.00001\}$. The following table summarizes the results from these trials.

C	ϵ	*accuracy*
0.1	0.001	0.264
0.01	0.001	0.397
0.001	0.001	0.325
0.1	**0.0001**	**0.633**
0.01	0.0001	0.572
0.001	0.00001	0.436
0.1	0.00001	0.385
0.01	0.00001	0.535
0.001	0.00001	0.242

The results above indicate that $C = 0.01$ and $\epsilon = 0.0001$ were appropriate values for SVM w/ SGD.

B. Custom Multi-Layer Neural Network (NN)

1. Background

A two-layer neural network was originally the first method of choice for this attempt in x-ray image classification, as it quickly became clear how difficult it'd be to kernelize a linear separator to optimally classify images into one of the four classes in $\mathcal{Y}$. This method involved creating a custom multi-layer neural network from scratch while recycling multiple steps from the SVM attempt, such as image matrix flattening, k-Fold Cross Validation, and validation using images exposed to every pre-processing method. Instead of the *one-against-all* technique from the previous procedure, the neural network incorporated a *softmax* activation layer at its output. Softmax works similarly to *one-against-all* in that it answers each of the "Is it..." questions with a value in $[0, +1]$. It does so, however, by taking the logit score for each outputted label and running it through the softmax function as follows to get an associated probability.[5]

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{3.4}$$

2. Algorithm

The neural network was developed using a sequential approach. Note that we will assume, for this algorithm, that the images were pre-preprocessed using Method A, resulting in heights and widths of 88 pixels for the inputs.

Since this neural network consumed images as one-dimensional vectors, each matrix of pixels had to be flattened into vectors of length $88 \times 88 = 7744$ and normalized by dividing by 255 (pixels are 1 byte in size). A training matrix X_{train} was then created from 80% of the originally labeled training data and the respective labels for each data point in X_{train} were simultaneously stored in Y_{train}. In order for the neural network to classify data points into any of the classes in $\mathcal{Y}$, the labels in Y_{train} were subsequently one-hot encoded to produce new outputs $[1, 0, 0, 0]$ for *normal*, $[0, 1, 0, 0]$ for *bacterial*, $[0, 0, 1, 0]$ for *viral*, and $[0, 0, 0, 1]$ for *covid*.

Evidently, the final layer of the neural network contained 4 neurons containing the outputs of the softmax function described earlier. The difficult task pertained to the research and trial and error involved with configuring the intermediary densely connected layers (or hidden layers) that connected the 7744-neuron-length input to the 4-neuron-length output.

Despite the neural network being configured to accept any number of hidden layers and any activation functions (which was obviously overkill), it was decided to use a simple hidden layer schematic and run the net on a large magnitude of epochs. The two hidden layers that were constructed consisted of a customary 512 neurons each and activated via the sigmoid function, as follows:

$$\text{Sigmoid}(x) = \frac{1}{1 + e^{-x}} \tag{3.5}$$

Using the sigmoid function for activation allowed us to keep negative outputs from each neuron really small, but still have them impact the weights of each connection during learning.

To learn the weights between each layer, the model was trained over 500 epochs with a learning rate of $l = 0.0001$ (the rate at which weights w and biases b are updated). For each epoch, in batch sizes of 32 data points, each batch of training data $X_{\text{train},i}$ was fed-forward through the neural network. At each layer L_j, the output $z_j = w_j^T x$ for $x \in X_{\text{train},i}$ and activated output $a_j = \text{Activate}(z_j)$ (based on the activation function at the current layer) were recorded. The final output vectors $Z =< z_1, z_2, ..., z_j, ..., z_n >$ and $A =< a_1, a_2, ..., a_j, ..., a_n >$ were then used as inputs to the backpropogation algorithm.

The cost function C used in backpropagation was Mean-Squared Error (MSE):

$$C(X, \theta) = \frac{1}{2N} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2, \tag{3.6}$$

where $\hat{y}_i$ is the calculated label and y_i is the true label.[6]. The error term for the output layer was calculated to be

$$\delta_{\text{output}} = \Delta\text{Softmax}(a_{\text{output}})(\hat{y}_{\text{c}} - y_{\text{c}}) \tag{3.7}$$

for a one-hot-encoded class c and gradient of the softmax activation function ΔSoftmax. Moreover, the error terms for the hidden layers were found to be

$$\delta_j = \Delta\text{Sigmoid}(a_{\text{output}}) \sum_{l=1}^{|L|-1} w_{jl} \delta_{j(l+1)}. \tag{3.8}$$

The value of the error term at each layer was thereby multiplied by the weights and biases of each term, and finally $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ were outputted from the backpropagation function. Finally, the weights and biases were updated at the end of each training batch as follows:

$$w \leftarrow w + l \frac{\partial C}{\partial w} \tag{3.9}$$

$$b \leftarrow b + l \frac{\partial C}{\partial b} \tag{3.10}$$

where l is the learning rate.

To make a prediction, a testing matrix X_{test} was fed forward using the algorithm described earlier and its output was one-hot-decoded at the last layer.

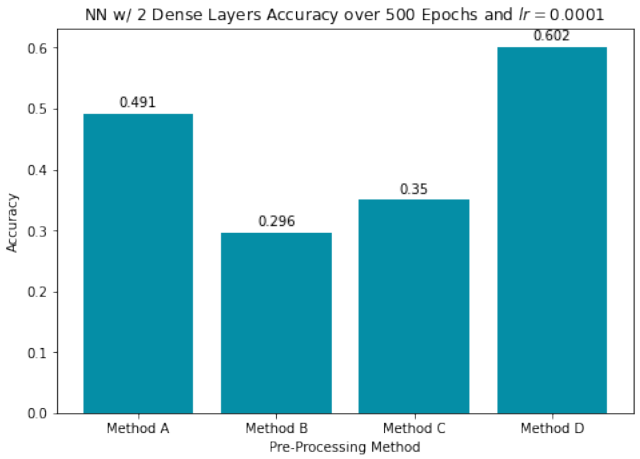


FIG. 5: Results for NN with k-Fold CV

3. *Results*

Figure 5 shows the results of *k-Fold Cross Validation* with $k = 10$ for each of the image pre-processing methods.

Figure 6 shows the confusion matrices over a holdout set of the same 225 randomly chosen training data points for each of the aforementioned methods.

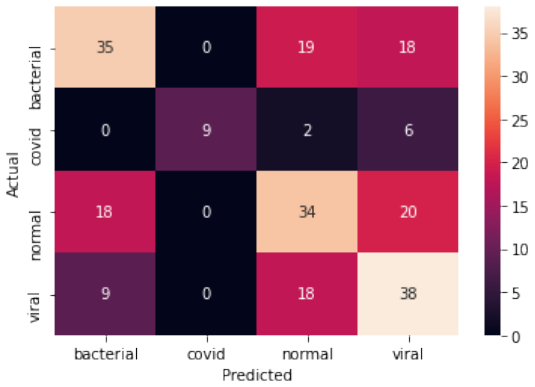
Once again, none of these results fared better than the minimum threshold accuracy of 0.80065, and so a final method for classification was examined afterwards.

4. *Commentary*

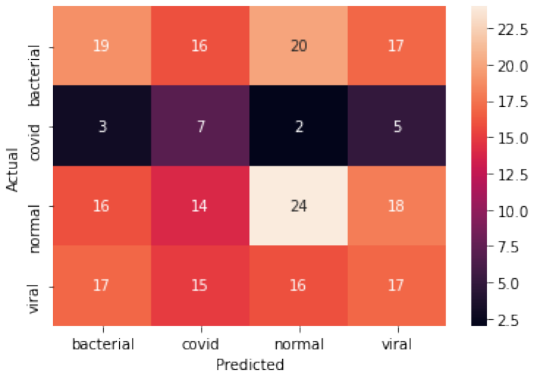
Unsurprisingly, the two-layer neural network chosen for this problem performed rather poorly on this dataset, considering it had to perform multi-class, not binary, classification onto $\mathcal{Y}$. The biggest mistake was likely choosing such a meager amount of neurons in each hidden layer (512) in relation to the input size of at least 7744, which could have resulted in data loss. Additionally, since every layer is densely connected, the spatial relationship between each adjacent pixel was not learned very well.

The best performing method here was, once again, Method D. This can be attributed to its compactness, as opposed to the outputs of Methods B and C, and the greater meaningfulness of each pixel in the vector (as opposed to outputs of method A that are framed by black pixels). This time, Method A outperformed the remaining two methods, but was unable to reach the 50% mark.

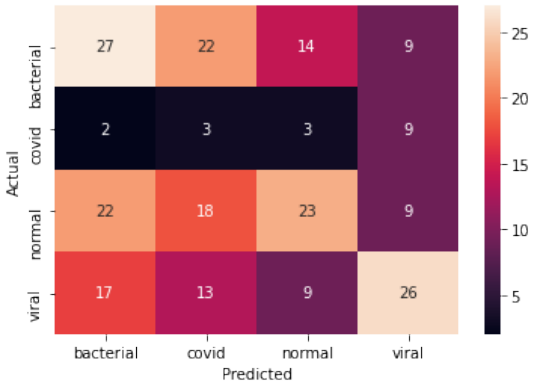
The confusion matrices in Figure 6 show very little correlation with one another, hinting that the neural network was struggling to classify each data point over only 500 epochs. While classifying data points from method A, it correctly predicted the most amount of *viral* data and predicted the least amount of *covid* data. The neural network excelled with *bacterial* data points for Meth-



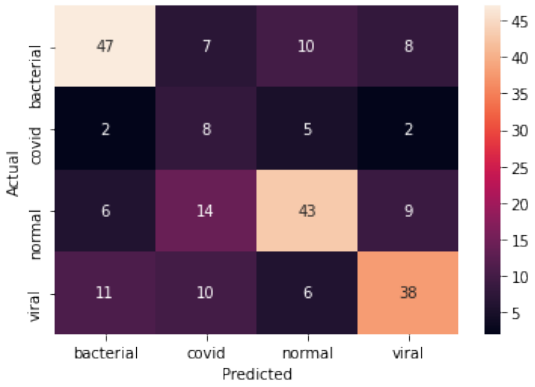
(a) Neural Net w/ Method A Pre-Processing



(b) Neural Net w/ Method B Pre-Processing



(c) Neural Net w/ Method C Pre-Processing



(d) Neural Net w/ Method D Pre-Processing

FIG. 6: Confusion matrices for classification of 255 data points over 500 epochs with a learning rate of $l = 0.0001$.

ods C and D, which saw 27 and 47 correct classifications, respectively. Finally, the neural network was fairly accurate with classifying normal lungs on images preprocessed with Method B.

As we had still not seen major improvements compared to the SVM method of classification, we propose a third method of classification in the following section.

C. Deep Residual Convolutional Neural Network (CNN) – CORONet

1. Background

After the two aforementioned classification methods did not achieve the threshold accuracy of 0.80065*, I knew it was time to bring in the big guns.*

For the third method, a CNN was constructed using Keras[7] and trained for 200 epochs on an augmented data set.

2. Algorithm

After researching how others trained CNNs to classify x-ray images, I was inspired by Adrian Rosebrock[8] to formulate my model in a similar structure to Kaiming He et. al.'s ResNet.[9] To abide by the hip trend of naming CNNs, this model was named CORONet, in reference to COVID-19 and in honor of Koronet Pizza on Broadway and 111th.[10]

For completeness, the layer summary for CORONet is shown in Figure 14. Nonetheless, a rough summary is provided as follows assuming Method A was used for pre-processing.

An input shape of $(200, 200, 1)$ was defined, where the first two sizes denoted the dimensions of the input image and the third dimension was a scalar denoting the grayscale value of the image. This is much unlike the original ResNet, which holds values for *red*, *green*, and *blue* in the third dimension. Next, a `ZeroPadding` layer is applied to reduce data loss from padding the image during convolution. Next, the image was passed through the first "convolutional group," where it was convoluted using a size 32-filter with a 9×9 kernel, batch normalized[11], activated using leaky RELU, and then max-pooled with pool dimensions of 3×3.[12] Convolution is the simple process of applying a filter to the input. Batch normalization applies a standard normal transformation onto the previous layer to speed up learning. The leaky RELU, as opposed to the Sigmoid and Softmax activations, leaves positive inputs unchanged but adds a fractional gradient on the negative end to enforce a penalty.[13] All instances of leaky RELU in this paper use a leak constant of 0.3. Lastly, max-pooling downsizes the input into the next layer by taking the largest value in each pool and outputting it into a smaller matrix. The process of max-pooling is shown in Figure 8.

Next, two types of blocks were defined–*Convolutional Blocks* and *Bottleneck Blocks*–which contain several layers and a shortcut. Before proceeding to define a block, we must define how a shortcut works in residual neural networks. Traditionally, neural networks and, explicitly, convolutional neural networks suffer degradation: the inverse relationship between network depth and testing accuracy. Shortcuts, which allow a layer to feed its output to a layer that is not directly subsequent to it, mitigate degradation by providing an extra term to the network, if we think of it as a mapping function. This allows for residual neural networks to be more easily optimized.[14] The process of shortcutting is shown in Figure 9.

Bottleneck blocks (Bottlenecks) begin by creating a shortcut to the layer in front of the block. Then, the same convolution-to-batch normalization-to-leaky RELU sequence is performed over three pre-defined filter sizes (i.e. 32, 64, and 128). Finally, the shortcut from earlier is reconnected to the CNN and the block is yet again activated via leaky RELU. These blocks are called "Bottleneck" blocks because the kernel sizes of the first and third convolutions are always 1×1, whereas the middle size is usually (but not always) 3×3. This theoretically decreases training loss by compacting more features into less space.

Convolution blocks (ConvBlocks) are very similar to Bottlenecks. Once again, they start with a shortcut that connects to the end of the block. The shortcut is followed by two convolution-to-batch normalization-to-leaky RELU sequences: the first with a 1×1 kernel size and the second with a greater-than-one filter size. It ends with a convolution-to-batch normalization sequence with another 1×1 filter size, to make three convolutions in total. This time, the shortcut is taken at the end and both convoluted and batch normalized (using the third filter size), before a final activation via leaky RELU. In case most of this sounds like gibberish, the blocks are visualized in Figure 7.

The middle groups of CORONet contained four groups of one ConvBlock and between two and five subsequent Bottlenecks. The goal was to reduce computation time and extract the most amount of meaningful features from the image. This is further discussed in the Commentary section.

The output of all blocks, with reconnected shortcuts, was first averagely-pooled (like max-pooling, but taking the average of each pool), and then flattened into a one dimensional vector. This vector was then connected to a dense network of 1024 neurons and activated via leaky RELU. Finally, a Dropout layer disconnected 50% of the weights from the previous layer to reduce overfitting, and fed the outputs into another densely connected layer of just four neurons activated via softmax: the exact same output layer we used in the second classification method.

The model starts with a learning rate of 0.1, and then uses the Keras callback `ReduceLROnPlateau`, which de-

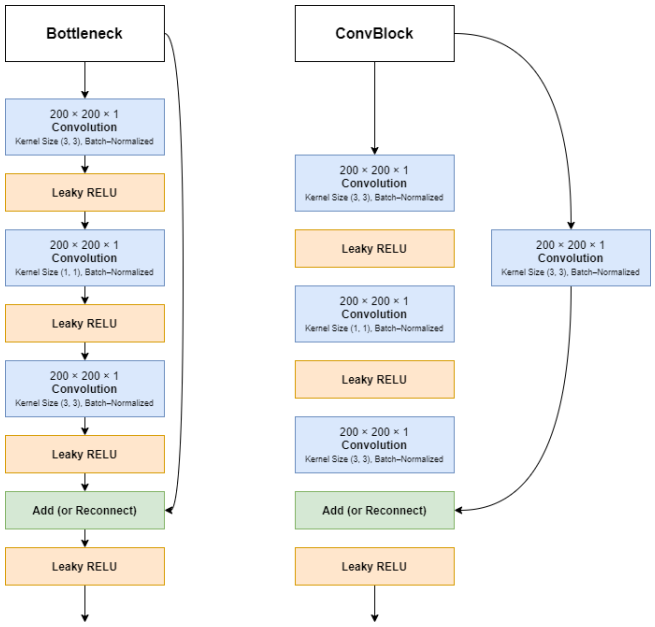


FIG. 7: Bottleneck Block and Convolutional Block, respectively.

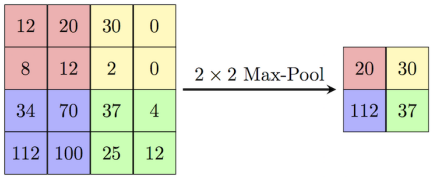


FIG. 8: Visualization of max-pooling with pool dimensions of 2×2.[16]

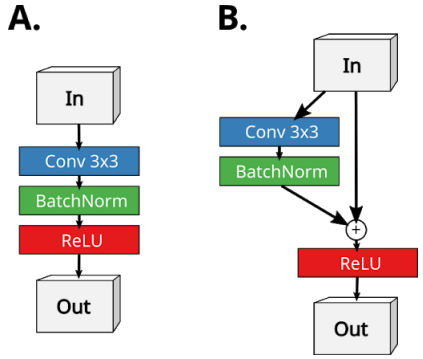


FIG. 9: Visualization of shortcutting in reference to He et. al.'s work.[17]

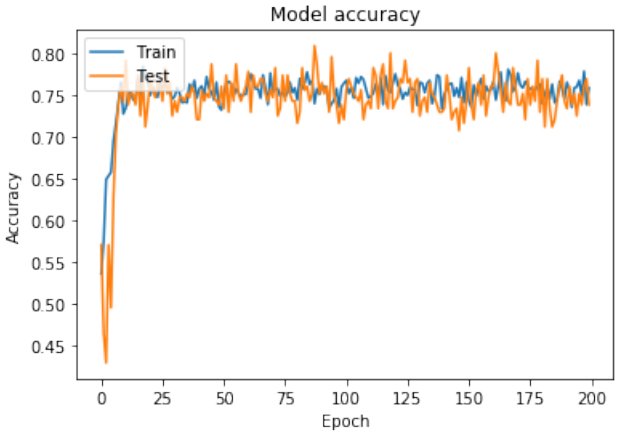


FIG. 10: First 200 epochs for CNN with densely connected layer of size 256.

creases the learning rate by a factor of 10 every time the validation loss remains within a small value $\epsilon = 0.1$ over at least 3 epochs. For this validation, the training data was split, once again, into a randomly selected 80/20 split, and validated automatically over each epoch in Keras.[15].

A custom callback `SaveBest` was created to track both the validation accuracy and the validation loss. When the loss reached a new minimum and the accuracy reached a new maximum, the model was then saved to the disk. That way, after training, the best version of the model was used instead of the version that had most recently been trained just by loading the model weights from a file.

3. *Results*

While experimenting classification of images from all four pre-processing methods, it became apparent that Method A excelled above the rest for a preset input size of 200×200. This is due to several factors:

1. Methods C and D cropped the images, and thus lost potentially valuable information.

2. Method B stretched one side to 200 pixels and the other side proportional to the ratio between the mean height and width of all images. This stretching caused important features to have skewed sizes.

3. CORONet effectively discarded information on useless features, such as the black borders around Method A, due to max-pooling at each layer and bottlenecking. It also gained important information on pixels adjacent to each other within each convolution layer's kernel.

Thus, analysis for this section will be solely based on Method A, as it had been the only method thoroughly tested.

The first attempt used Method A images with a second-to-last densely connected layer containing 256 neurons. The CNN averaged roughly 80% accuracy on 225 randomly chosen training data points, leading to a score of 0.81372 on the Kaggle submission. The training and testing (validation) accuracies are shown in Figure 10.

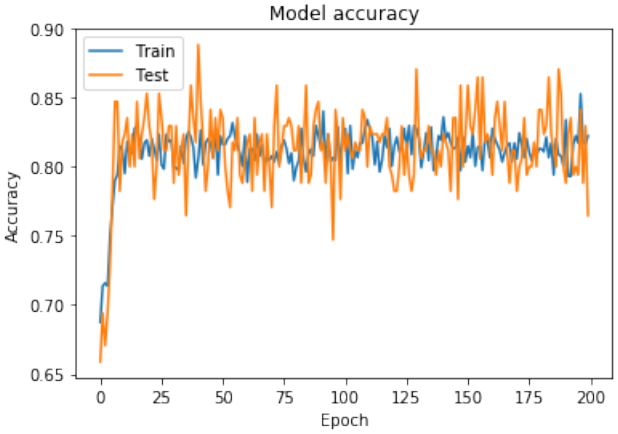


FIG. 11: First 200 epochs for CNN with densely connected layer of size 256 and enhanced image augmentation.

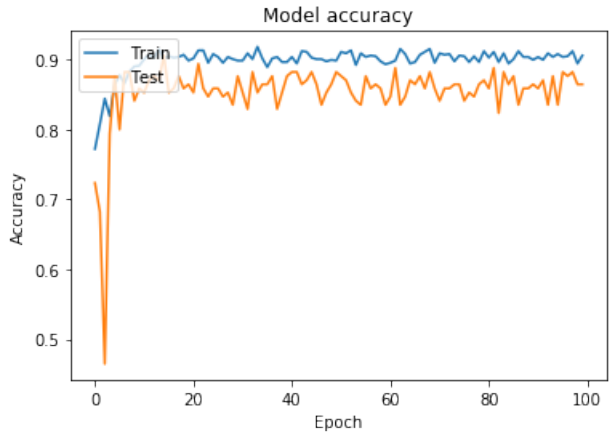


FIG. 12: First 200 epochs for CNN with densely connected layer of size 1024 and enhanced image augmentation.

A slight improvement was made when the rotation range of the augmented images was changed from 10 degrees to 20 degrees and all other shifting and scaling ranges were bumped from a 0.2 scale to a 0.3 scale. This way, the model was effectively exposed to more variation in the training data. The CNN attained a maximum accuracy of 81% on the same data points, leading to a score of 0.84640 on the Kaggle submission. The training and testing accuracies are shown in Figure 11.

Finally, the number of neurons in the final layer were increased to 1024 and the CNN was trained over the same training data. The CNN saw a maximum accuracy of 89% and the Kaggle submission scored 0.87908, indicating some overfitting around the testing data. The training and testing accuracies are shown in Figure 12.

A confusion matrix for the 225 predicted data points using the final, most accurate CNN configuration is shown in Figure 13. This confusion matrix shows the

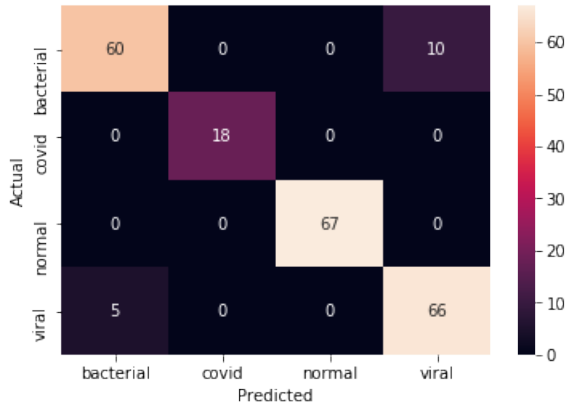


FIG. 13: Confusing matrix for CNN trained on an enhanced, augmented data set over 200 epochs.

predictions on the same holdout validation data, resulting in a score of 0.9336283185840708.

The confusion matrix highlights this classifier's success over the SVM and simple NN, where the only errors it makes are distinguishing between *bacterial* and *viral* images. This agrees with the original heuristic examination of the training data made at the beginning of this paper, in which it was stated that "Bacterial lungs look very similar to viral lungs." It appears machines have some difficulty spotting the difference as well.

4. Commentary

The greatest obstacle in training CORONet was the aniticipated lack of training data. With only 1,127 training images in total (which also needed to be split for validation), it was, at first, difficult to prevent low accuracy and blatant overfitting.

Then, instead of training directly on the training images, Keras's ImageDataGenerator[18] class was utilized to augment the training data through rotation, horizontal flipping, zooming, shearing, and shifting. Whereas this did not directly "add" training data, it allowed the model to train on an increased number of x-ray scenarios, thus greatly contributing to the number of different scenarios faced by the CNN.

IV. CONNECTING TO THE REAL WORLD

Whereas the SVM and NN classifier failed to meet the classification threshold of 0.80065, CORONet's classification prowess emphasized by a Kaggle score of 0.87908 leads to speculation on whether or not it may be plausible to use in real-world scenarios.

If CORONet were to be used for classification in a clinical setting, it would thus be imperative for medical practitioners to make final "judgements" on whether or not

lungs are bacterially or virally infected. This nullifies the automation provided by the classifier in the diagnosing process, and may thus be a reason not to perform automatic diagnoses. In most cases, however, the most important judgement is whether or not a patient is infected at all and, in this case, the classifier excels. Additionally, it excels in identifying cases of COVID-19, and, at least on the basis of its performance on the holdout training data set, there is much reason to use this classifier during this crisis..

The final discussion concerns the *interpretability* of the classifiers used in real-world scenarios. As stated in the beginning of the paper, x-ray image classification depends on extracting patterns among numerous adjacent pixels. The SVM and NN seemingly fail to draw conclusions on these patterns, instead relying on colors and positions of each individual pixel. This leads to difficulties in classifying off-centered and rotated images that clinicians often have no problem discerning. Through kernelization, CORONet is able to extract parts of the image and make decisions on a smaller scale, while discarding information it deems less relevant to classification. This, combined with training on augmented image data that contains flipped and rotated images, makes CORONet's decision-making process resemble that of a real practitioner. While it's hard to truly discern which features makes a real difference in the classification of each image, this study has shown that well-trained CNNs can be impactful in diagnosing COVID-19 and other lung infections.

V. REFERENCES

[1] "COVID-19, MERS SARS." *National Institute of Allergy and Infectious Diseases*, U.S. Department of Health and Human Services, https://www.niaid.nih.gov/diseases-conditions/covid-19.

[2] "Image Pre-Processing for Chest X-Ray." *Kaggle*, Kaggle, 26 June 2019, https://www.kaggle.com/seriousran/image-pre-processing-for-chest-x-ray.

[3] Leeds, Daniel. D., "CISC 5800: Machine Learning." *Fordham University*, 28 April 2020, https://storm.cis.fordham.edu/~leeds/cisc5800/.

[4] Vieira, Bruno Hebling. "Meaning of Epsilon in SVM regression", *Stack Exchange*, https://stats.stackexchange.com/questions/259018/meaning-of-epsilon-in-svm-regression.

[5] Bridle, J.S. "Probabilistic Interpretation of Feedforward Classification Network Outputs, with Relationships to Statistical Pattern Recognition", *Soulié F.F., Hérault J. (eds) Neurocomputing*, NATO ASI Series (Series F: Computer and Systems Sciences), vol 68. Springer, Berlin, Heidelberg.

[6] "Backpropagation." *Brilliant Math Science Wiki*, brilliant.org/wiki/backpropagation/.

[7] Chollet, François and others. Keras, 2015, https://keras.io.

[8] Rosebrock, Adrian. "Detecting COVID-19 in X-Ray Images with Keras, TensorFlow, and Deep Learning." *PyImageSearch*, 18 Apr. 2020, https://www.pyimagesearch.com/2020/03/16/detecting-covid-19-in-x-ray-images-with-keras-tensorflow-a

[9] He, Kaiming, et al. "Deep Residual Learning for Image Recognition." *ArXiv.org*, 10 Dec. 2015, https://arxiv.org/abs/1512.03385.

[10] Koronet Pizza. *Google Maps*. https://goo.gl/maps/yJQaNhbcuogESE7C7.

[11] Ioffe, Sergey, and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." *ArXiv.org*, 2 Mar. 2015, https://arxiv.org/abs/1502.03167.

[12] Chollet, François and others. Keras, 2015, https://keras.io/layers/convolutional.

[13] Xu, Bing, et al. "Empirical Evaluation of Rectified Activations in Convolutional Network." *ArXiv.org*, 5 May. 2015, https://arxiv.org/abs/1505.00853.

[14] Liu, Tianyi, et al. "Towards Understanding the Importance of Shortcut Connections in Residual Networks." *ArXiv.org*, 2 Nov. 2019, https://arxiv.org/abs/1909.04653.

[15] Chollet, François and others. Keras, 2015, https://keras.io/layers/callbacks

[16] "Max-Pooling / Pooling." *Computer Science Wiki*, https://computersciencewiki.org/index.php/Max-pooling_/_Pooling.

[17] Williford, Jonathan R. "Review of He Et Al. 2015 *Deep Residual Learning for Image Recognition*." *Neural Vision*, 5 June 2017, neural.vision/blog/article-reviews/deep-learning/he-resnet-2015/.

[18] Chollet, François and others. Keras, 2015, https://keras.io/preprocessing/image

Layer (type)	Output Shape	Param #	Connected to
input_2 (InputLayer)	(None, 200, 200, 1)	0	
zero_padding2d_1 (ZeroPadding2D	(None, 206, 206, 1)	0	input_2[0][0]
conv2d_1 (Conv2D)	(None, 206, 206, 32)	2624	zero_padding2d_1[0][0]
batch_normalization_1 (BatchNor	(None, 206, 206, 32)	128	conv2d_1[0][0]
leaky_re_lu_1 (LeakyReLU)	(None, 206, 206, 32)	0	batch_normalization_1[0][0]
max_pooling2d_2 (MaxPooling2D)	(None, 68, 68, 32)	0	leaky_re_lu_1[0][0]
conv2d_2 (Conv2D)	(None, 68, 68, 32)	1056	max_pooling2d_2[0][0]
batch_normalization_2 (BatchNor	(None, 68, 68, 32)	128	conv2d_2[0][0]
leaky_re_lu_2 (LeakyReLU)	(None, 68, 68, 32)	0	batch_normalization_2[0][0]
conv2d_3 (Conv2D)	(None, 68, 68, 32)	9248	leaky_re_lu_2[0][0]
batch_normalization_3 (BatchNor	(None, 68, 68, 32)	128	conv2d_3[0][0]
leaky_re_lu_3 (LeakyReLU)	(None, 68, 68, 32)	0	batch_normalization_3[0][0]
conv2d_4 (Conv2D)	(None, 68, 68, 128)	4224	leaky_re_lu_3[0][0]
conv2d_5 (Conv2D)	(None, 68, 68, 128)	4224	max_pooling2d_2[0][0]
batch_normalization_4 (BatchNor	(None, 68, 68, 128)	512	conv2d_4[0][0]
batch_normalization_5 (BatchNor	(None, 68, 68, 128)	512	conv2d_5[0][0]
add_17 (Add)	(None, 68, 68, 128)	0	batch_normalization_4[0][0] batch_normalization_5[0][0]
leaky_re_lu_4 (LeakyReLU)	(None, 68, 68, 128)	0	add_17[0][0]
conv2d_6 (Conv2D)	(None, 68, 68, 32)	4128	leaky_re_lu_4[0][0]
batch_normalization_6 (BatchNor	(None, 68, 68, 32)	128	conv2d_6[0][0]
leaky_re_lu_5 (LeakyReLU)	(None, 68, 68, 32)	0	batch_normalization_6[0][0]
conv2d_7 (Conv2D)	(None, 68, 68, 32)	9248	leaky_re_lu_5[0][0]
batch_normalization_7 (BatchNor	(None, 68, 68, 32)	128	conv2d_7[0][0]
leaky_re_lu_6 (LeakyReLU)	(None, 68, 68, 32)	0	batch_normalization_7[0][0]
conv2d_8 (Conv2D)	(None, 68, 68, 128)	4224	leaky_re_lu_6[0][0]
batch_normalization_8 (BatchNor	(None, 68, 68, 128)	512	conv2d_8[0][0]
leaky_re_lu_7 (LeakyReLU)	(None, 68, 68, 128)	0	batch_normalization_8[0][0]
add_18 (Add)	(None, 68, 68, 128)	0	leaky_re_lu_7[0][0] leaky_re_lu_4[0][0]
leaky_re_lu_8 (LeakyReLU)	(None, 68, 68, 128)	0	add_18[0][0]
conv2d_9 (Conv2D)	(None, 68, 68, 32)	4128	leaky_re_lu_8[0][0]
batch_normalization_9 (BatchNor	(None, 68, 68, 32)	128	conv2d_9[0][0]
leaky_re_lu_9 (LeakyReLU)	(None, 68, 68, 32)	0	batch_normalization_9[0][0]
conv2d_10 (Conv2D)	(None, 68, 68, 32)	9248	leaky_re_lu_9[0][0]

batch_normalization_10 (BatchNo	(None, 68, 68, 32)	128	conv2d_10[0][0]
leaky_re_lu_10 (LeakyReLU)	(None, 68, 68, 32)	0	batch_normalization_10[0][0]
conv2d_11 (Conv2D)	(None, 68, 68, 128)	4224	leaky_re_lu_10[0][0]
batch_normalization_11 (BatchNo	(None, 68, 68, 128)	512	conv2d_11[0][0]
leaky_re_lu_11 (LeakyReLU)	(None, 68, 68, 128)	0	batch_normalization_11[0][0]
add_19 (Add)	(None, 68, 68, 128)	0	leaky_re_lu_11[0][0] leaky_re_lu_8[0][0]
leaky_re_lu_12 (LeakyReLU)	(None, 68, 68, 128)	0	add_19[0][0]
conv2d_12 (Conv2D)	(None, 34, 34, 64)	8256	leaky_re_lu_12[0][0]
batch_normalization_12 (BatchNo	(None, 34, 34, 64)	256	conv2d_12[0][0]
leaky_re_lu_13 (LeakyReLU)	(None, 34, 34, 64)	0	batch_normalization_12[0][0]
conv2d_13 (Conv2D)	(None, 34, 34, 64)	36928	leaky_re_lu_13[0][0]
batch_normalization_13 (BatchNo	(None, 34, 34, 64)	256	conv2d_13[0][0]
leaky_re_lu_14 (LeakyReLU)	(None, 34, 34, 64)	0	batch_normalization_13[0][0]
conv2d_14 (Conv2D)	(None, 34, 34, 256)	16640	leaky_re_lu_14[0][0]
conv2d_15 (Conv2D)	(None, 34, 34, 256)	33024	leaky_re_lu_12[0][0]
batch_normalization_14 (BatchNo	(None, 34, 34, 256)	1024	conv2d_14[0][0]
batch_normalization_15 (BatchNo	(None, 34, 34, 256)	1024	conv2d_15[0][0]
add_20 (Add)	(None, 34, 34, 256)	0	batch_normalization_14[0][0] batch_normalization_15[0][0]
leaky_re_lu_15 (LeakyReLU)	(None, 34, 34, 256)	0	add_20[0][0]
conv2d_16 (Conv2D)	(None, 34, 34, 64)	16448	leaky_re_lu_15[0][0]
batch_normalization_16 (BatchNo	(None, 34, 34, 64)	256	conv2d_16[0][0]
leaky_re_lu_16 (LeakyReLU)	(None, 34, 34, 64)	0	batch_normalization_16[0][0]
conv2d_17 (Conv2D)	(None, 34, 34, 64)	36928	leaky_re_lu_16[0][0]
batch_normalization_17 (BatchNo	(None, 34, 34, 64)	256	conv2d_17[0][0]
leaky_re_lu_17 (LeakyReLU)	(None, 34, 34, 64)	0	batch_normalization_17[0][0]
conv2d_18 (Conv2D)	(None, 34, 34, 256)	16640	leaky_re_lu_17[0][0]
batch_normalization_18 (BatchNo	(None, 34, 34, 256)	1024	conv2d_18[0][0]
leaky_re_lu_18 (LeakyReLU)	(None, 34, 34, 256)	0	batch_normalization_18[0][0]
add_21 (Add)	(None, 34, 34, 256)	0	leaky_re_lu_18[0][0] leaky_re_lu_15[0][0]
leaky_re_lu_19 (LeakyReLU)	(None, 34, 34, 256)	0	add_21[0][0]
conv2d_19 (Conv2D)	(None, 34, 34, 64)	16448	leaky_re_lu_19[0][0]
batch_normalization_19 (BatchNo	(None, 34, 34, 64)	256	conv2d_19[0][0]
leaky_re_lu_20 (LeakyReLU)	(None, 34, 34, 64)	0	batch_normalization_19[0][0]
conv2d_20 (Conv2D)	(None, 34, 34, 64)	36928	leaky_re_lu_20[0][0]

```
batch_normalization_20 (BatchNo (None, 34, 34, 64)   256         conv2d_20[0][0]
leaky_re_lu_21 (LeakyReLU)      (None, 34, 34, 64)   0           batch_normalization_20[0][0]
conv2d_21 (Conv2D)              (None, 34, 34, 256)  16640       leaky_re_lu_21[0][0]
batch_normalization_21 (BatchNo (None, 34, 34, 256)  1024        conv2d_21[0][0]
leaky_re_lu_22 (LeakyReLU)      (None, 34, 34, 256)  0           batch_normalization_21[0][0]
add_22 (Add)                    (None, 34, 34, 256)  0           leaky_re_lu_22[0][0]
                                                                 leaky_re_lu_19[0][0]
leaky_re_lu_23 (LeakyReLU)      (None, 34, 34, 256)  0           add_22[0][0]
conv2d_22 (Conv2D)              (None, 17, 17, 128)  32896       leaky_re_lu_23[0][0]
batch_normalization_22 (BatchNo (None, 17, 17, 128)  512         conv2d_22[0][0]
leaky_re_lu_24 (LeakyReLU)      (None, 17, 17, 128)  0           batch_normalization_22[0][0]
conv2d_23 (Conv2D)              (None, 17, 17, 128)  147584      leaky_re_lu_24[0][0]
batch_normalization_23 (BatchNo (None, 17, 17, 128)  512         conv2d_23[0][0]
leaky_re_lu_25 (LeakyReLU)      (None, 17, 17, 128)  0           batch_normalization_23[0][0]
conv2d_24 (Conv2D)              (None, 17, 17, 512)  66048       leaky_re_lu_25[0][0]
conv2d_25 (Conv2D)              (None, 17, 17, 512)  131584      leaky_re_lu_23[0][0]
batch_normalization_24 (BatchNo (None, 17, 17, 512)  2048        conv2d_24[0][0]
batch_normalization_25 (BatchNo (None, 17, 17, 512)  2048        conv2d_25[0][0]
add_23 (Add)                    (None, 17, 17, 512)  0           batch_normalization_24[0][0]
                                                                 batch_normalization_25[0][0]
leaky_re_lu_26 (LeakyReLU)      (None, 17, 17, 512)  0           add_23[0][0]
conv2d_26 (Conv2D)              (None, 17, 17, 128)  65664       leaky_re_lu_26[0][0]
batch_normalization_26 (BatchNo (None, 17, 17, 128)  512         conv2d_26[0][0]
leaky_re_lu_27 (LeakyReLU)      (None, 17, 17, 128)  0           batch_normalization_26[0][0]
conv2d_27 (Conv2D)              (None, 17, 17, 128)  147584      leaky_re_lu_27[0][0]
batch_normalization_27 (BatchNo (None, 17, 17, 128)  512         conv2d_27[0][0]
leaky_re_lu_28 (LeakyReLU)      (None, 17, 17, 128)  0           batch_normalization_27[0][0]
conv2d_28 (Conv2D)              (None, 17, 17, 512)  66048       leaky_re_lu_28[0][0]
batch_normalization_28 (BatchNo (None, 17, 17, 512)  2048        conv2d_28[0][0]
leaky_re_lu_29 (LeakyReLU)      (None, 17, 17, 512)  0           batch_normalization_28[0][0]
add_24 (Add)                    (None, 17, 17, 512)  0           leaky_re_lu_29[0][0]
                                                                 leaky_re_lu_26[0][0]
leaky_re_lu_30 (LeakyReLU)      (None, 17, 17, 512)  0           add_24[0][0]
conv2d_29 (Conv2D)              (None, 17, 17, 128)  65664       leaky_re_lu_30[0][0]
batch_normalization_29 (BatchNo (None, 17, 17, 128)  512         conv2d_29[0][0]
leaky_re_lu_31 (LeakyReLU)      (None, 17, 17, 128)  0           batch_normalization_29[0][0]
conv2d_30 (Conv2D)              (None, 17, 17, 128)  147584      leaky_re_lu_31[0][0]
```

batch_normalization_30 (BatchNo	(None, 17, 17, 128)	512	conv2d_30[0][0]
leaky_re_lu_32 (LeakyReLU)	(None, 17, 17, 128)	0	batch_normalization_30[0][0]
conv2d_31 (Conv2D)	(None, 17, 17, 512)	66048	leaky_re_lu_32[0][0]
batch_normalization_31 (BatchNo	(None, 17, 17, 512)	2048	conv2d_31[0][0]
leaky_re_lu_33 (LeakyReLU)	(None, 17, 17, 512)	0	batch_normalization_31[0][0]
add_25 (Add)	(None, 17, 17, 512)	0	leaky_re_lu_33[0][0] leaky_re_lu_30[0][0]
leaky_re_lu_34 (LeakyReLU)	(None, 17, 17, 512)	0	add_25[0][0]
conv2d_32 (Conv2D)	(None, 17, 17, 128)	65664	leaky_re_lu_34[0][0]
batch_normalization_32 (BatchNo	(None, 17, 17, 128)	512	conv2d_32[0][0]
leaky_re_lu_35 (LeakyReLU)	(None, 17, 17, 128)	0	batch_normalization_32[0][0]
conv2d_33 (Conv2D)	(None, 17, 17, 128)	147584	leaky_re_lu_35[0][0]
batch_normalization_33 (BatchNo	(None, 17, 17, 128)	512	conv2d_33[0][0]
leaky_re_lu_36 (LeakyReLU)	(None, 17, 17, 128)	0	batch_normalization_33[0][0]
conv2d_34 (Conv2D)	(None, 17, 17, 512)	66048	leaky_re_lu_36[0][0]
batch_normalization_34 (BatchNo	(None, 17, 17, 512)	2048	conv2d_34[0][0]
leaky_re_lu_37 (LeakyReLU)	(None, 17, 17, 512)	0	batch_normalization_34[0][0]
add_26 (Add)	(None, 17, 17, 512)	0	leaky_re_lu_37[0][0] leaky_re_lu_34[0][0]
leaky_re_lu_38 (LeakyReLU)	(None, 17, 17, 512)	0	add_26[0][0]
conv2d_35 (Conv2D)	(None, 17, 17, 128)	65664	leaky_re_lu_38[0][0]
batch_normalization_35 (BatchNo	(None, 17, 17, 128)	512	conv2d_35[0][0]
leaky_re_lu_39 (LeakyReLU)	(None, 17, 17, 128)	0	batch_normalization_35[0][0]
conv2d_36 (Conv2D)	(None, 17, 17, 128)	147584	leaky_re_lu_39[0][0]
batch_normalization_36 (BatchNo	(None, 17, 17, 128)	512	conv2d_36[0][0]
leaky_re_lu_40 (LeakyReLU)	(None, 17, 17, 128)	0	batch_normalization_36[0][0]
conv2d_37 (Conv2D)	(None, 17, 17, 512)	66048	leaky_re_lu_40[0][0]
batch_normalization_37 (BatchNo	(None, 17, 17, 512)	2048	conv2d_37[0][0]
leaky_re_lu_41 (LeakyReLU)	(None, 17, 17, 512)	0	batch_normalization_37[0][0]
add_27 (Add)	(None, 17, 17, 512)	0	leaky_re_lu_41[0][0] leaky_re_lu_38[0][0]
leaky_re_lu_42 (LeakyReLU)	(None, 17, 17, 512)	0	add_27[0][0]
conv2d_38 (Conv2D)	(None, 17, 17, 128)	65664	leaky_re_lu_42[0][0]
batch_normalization_38 (BatchNo	(None, 17, 17, 128)	512	conv2d_38[0][0]
leaky_re_lu_43 (LeakyReLU)	(None, 17, 17, 128)	0	batch_normalization_38[0][0]

conv2d_39 (Conv2D)	(None, 17, 17, 128)	147584	leaky_re_lu_43[0][0]
batch_normalization_39 (BatchNo	(None, 17, 17, 128)	512	conv2d_39[0][0]
leaky_re_lu_44 (LeakyReLU)	(None, 17, 17, 128)	0	batch_normalization_39[0][0]
conv2d_40 (Conv2D)	(None, 17, 17, 512)	66048	leaky_re_lu_44[0][0]
batch_normalization_40 (BatchNo	(None, 17, 17, 512)	2048	conv2d_40[0][0]
leaky_re_lu_45 (LeakyReLU)	(None, 17, 17, 512)	0	batch_normalization_40[0][0]
add_28 (Add)	(None, 17, 17, 512)	0	leaky_re_lu_45[0][0] leaky_re_lu_42[0][0]
leaky_re_lu_46 (LeakyReLU)	(None, 17, 17, 512)	0	add_28[0][0]
conv2d_41 (Conv2D)	(None, 9, 9, 256)	131328	leaky_re_lu_46[0][0]
batch_normalization_41 (BatchNo	(None, 9, 9, 256)	1024	conv2d_41[0][0]
leaky_re_lu_47 (LeakyReLU)	(None, 9, 9, 256)	0	batch_normalization_41[0][0]
conv2d_42 (Conv2D)	(None, 9, 9, 256)	590080	leaky_re_lu_47[0][0]
batch_normalization_42 (BatchNo	(None, 9, 9, 256)	1024	conv2d_42[0][0]
leaky_re_lu_48 (LeakyReLU)	(None, 9, 9, 256)	0	batch_normalization_42[0][0]
conv2d_43 (Conv2D)	(None, 9, 9, 1024)	263168	leaky_re_lu_48[0][0]
conv2d_44 (Conv2D)	(None, 9, 9, 1024)	525312	leaky_re_lu_46[0][0]
batch_normalization_43 (BatchNo	(None, 9, 9, 1024)	4096	conv2d_43[0][0]
batch_normalization_44 (BatchNo	(None, 9, 9, 1024)	4096	conv2d_44[0][0]
add_29 (Add)	(None, 9, 9, 1024)	0	batch_normalization_43[0][0] batch_normalization_44[0][0]
leaky_re_lu_49 (LeakyReLU)	(None, 9, 9, 1024)	0	add_29[0][0]
conv2d_45 (Conv2D)	(None, 9, 9, 256)	262400	leaky_re_lu_49[0][0]
batch_normalization_45 (BatchNo	(None, 9, 9, 256)	1024	conv2d_45[0][0]
leaky_re_lu_50 (LeakyReLU)	(None, 9, 9, 256)	0	batch_normalization_45[0][0]
conv2d_46 (Conv2D)	(None, 9, 9, 256)	590080	leaky_re_lu_50[0][0]
batch_normalization_46 (BatchNo	(None, 9, 9, 256)	1024	conv2d_46[0][0]
leaky_re_lu_51 (LeakyReLU)	(None, 9, 9, 256)	0	batch_normalization_46[0][0]
conv2d_47 (Conv2D)	(None, 9, 9, 1024)	263168	leaky_re_lu_51[0][0]
batch_normalization_47 (BatchNo	(None, 9, 9, 1024)	4096	conv2d_47[0][0]
leaky_re_lu_52 (LeakyReLU)	(None, 9, 9, 1024)	0	batch_normalization_47[0][0]
add_30 (Add)	(None, 9, 9, 1024)	0	leaky_re_lu_52[0][0] leaky_re_lu_49[0][0]
leaky_re_lu_53 (LeakyReLU)	(None, 9, 9, 1024)	0	add_30[0][0]
conv2d_48 (Conv2D)	(None, 9, 9, 256)	262400	leaky_re_lu_53[0][0]

batch_normalization_48 (BatchNo	(None, 9, 9, 256)	1024	conv2d_48[0][0]
leaky_re_lu_54 (LeakyReLU)	(None, 9, 9, 256)	0	batch_normalization_48[0][0]
conv2d_49 (Conv2D)	(None, 9, 9, 256)	590080	leaky_re_lu_54[0][0]
batch_normalization_49 (BatchNo	(None, 9, 9, 256)	1024	conv2d_49[0][0]
leaky_re_lu_55 (LeakyReLU)	(None, 9, 9, 256)	0	batch_normalization_49[0][0]
conv2d_50 (Conv2D)	(None, 9, 9, 1024)	263168	leaky_re_lu_55[0][0]
batch_normalization_50 (BatchNo	(None, 9, 9, 1024)	4096	conv2d_50[0][0]
leaky_re_lu_56 (LeakyReLU)	(None, 9, 9, 1024)	0	batch_normalization_50[0][0]
add_31 (Add)	(None, 9, 9, 1024)	0	leaky_re_lu_56[0][0] leaky_re_lu_53[0][0]
leaky_re_lu_57 (LeakyReLU)	(None, 9, 9, 1024)	0	add_31[0][0]
average_pooling2d_1 (AveragePoo	(None, 4, 4, 1024)	0	leaky_re_lu_57[0][0]
flatten_1 (Flatten)	(None, 16384)	0	average_pooling2d_1[0][0]
dense_3 (Dense)	(None, 1024)	16778240	flatten_1[0][0]
leaky_re_lu_58 (LeakyReLU)	(None, 1024)	0	dense_3[0][0]
dropout_1 (Dropout)	(None, 1024)	0	leaky_re_lu_58[0][0]
dense_4 (Dense)	(None, 4)	4100	dropout_1[0][0]

FIG. 14: Design of the CORONet CNN for input pre-processed via Method A.