Deep COVID: An Exposition

Throughout our report, we present the contents of 'Deep-COVID: Predicting COVID-19 from chest X-ray images using deep transfer learning' (Deep-COVID, for short) by Minaee et al. We start with an introduction to the research question and overview of the data. Following this, we fit a logistic regression to the classification task. This will be used as our baseline model to compare the models used in the paper against. It also serves as an introduction to the basic concepts of Artificial Neural Networks (ANNs). Using this as a segue, we, next, discuss the basic concepts involved in designing, and training ANN's. Once this is done, we introduce the concepts particular to Convolutional Neural Networks (CNNs). Then, we briefly discuss transfer learning and present the models used in the paper. Finally, we review their model evaluation metrics in comparison with eachother, our initial regression model, and discuss potential future work.

The Goal of Deep-COVID and the Data

The Goal

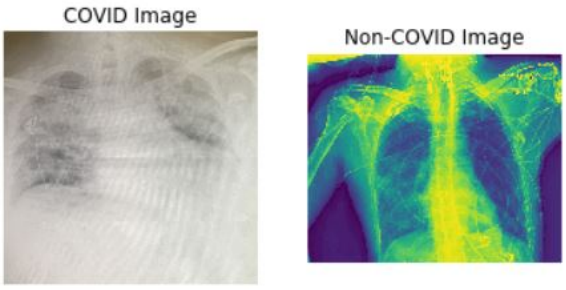
The COVID pandemic represents one of the biggest challenges currently facing the United States and the rest of the world. Before vaccination is sufficiently deployed and adopted, efforts towards treatment and prevention are of much importance. One of the most effective tools we have at our disposal is accurate and speedy detection. Diagnosing cases from radiography and radiology image is, perhaps, one of the fastest forms of detection (Minaee et al., 2020). In their paper, Minaee et al. apply transfer learning to fit famous CNNs, including ResNet18, ResNet50, SqueezeNet, and DenseNet-121, to identify COVID-19 disease in analyzed chest X-ray images. Results were promising and it is the hope that these methods can be used to more accurately and quickly identify cases in exposed individuals. We'll discuss, in more detail, the concepts of transfer learning, as well as the basic concepts necessary to understand how the ANNs used are structured. First, however, let's become more acquainted with the data used for training and testing.

The Data

The authors of Deep-COVID were nice enough to share a subset of their data along with a description of how it was collected, labeled, and organized in a Github repository (https://github.com/shervinmin/DeepCovid/tree/master/data) [https://github.com/shervinmin/DeepCovid/tree/master/data (https://github.com/shervinmin/DeepCovid/tree/master/data) [https://github.com/shervinmin/DeepCovid/tree/master/data)]. Still, we will give a brief synopsis in this section.

The data used by the authors comes from two sources:

- "Covid-Chestxray-Dataset", for COVID-positive samples
- "ChexPert Dataset", for COVID-negative samples



Sample images.

From the Covid-Chestxray-Dataset, only images that were selected to have a clear sign of COVID by the authors' board-certified radiologists were used: a total of 71 images, split 31 for training and 40 for testing. Several data-augmentation techniques (as well as over-sampling) were used to increase the number of COVID-19 samples, leading to final training/test split counts of:

Split	COVID-19	Non-COVID
Training Set	84 (420 after augmentation)	2000
Test Set	100	3000

Data train/test split summary (Minaee et al., 2020)

We note that one of the limitations of this project is the absence of a larger quantity of data, especially verified COVID-positive images. With the help of more data, we could hope for even better results. Now let's turn to our first model, a logistic regression model.

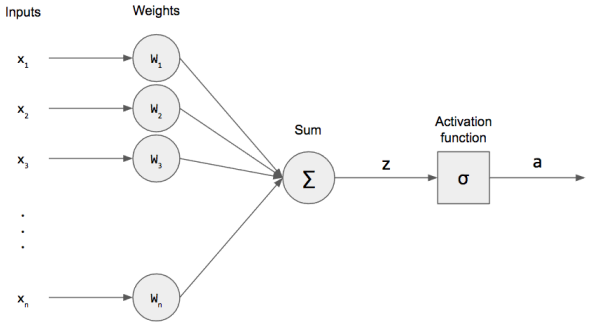
Logistic Regression: A Baseline Model and Segue into ANNs

Introduction and Defining the Model

In this section, we will fit a logitstic regression classifier to a subset of the author's data. This will serve two purposes. On the one hand, it will provide a baseline model to compare against the models used in Deep-COVID. And, on the other, it will serve as a segue into ANNs. Let us briefly recall the functional form of a logistic regression model. Namely, given a binary classification problem (say the two classes are 0 or 1) and a data point $\mathbf{x}$, the probability that $\mathbf{x}$ belongs to class 1 is given by:

$$P(y_i = 1|\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}^T\mathbf{x}}}.$$

This functional form can be thought of as the composition of two component forms. Namely, a linear component, $\mathbf{z}_i = \mathbf{w}^T\mathbf{x} = w_1x_1 + \ldots + w_px_p$ and a non-linear component, $a_i = \sigma(\mathbf{a}_i)$, where $y_i = a_i$ and $\sigma(x) = \frac{1}{1+e^{-x}}$ denotes a sigmoid function, which we will refer to as the sigmoid *activation* function. Thus, graphically, we can think of our model as follows:



Perceptron architecture (Deshpande 2020)

In order to train a model of this form to our classification task, we need to define how our input layer will look, i.e. what are our features. In the case of image classification tasks, it is common to use a matrix (2 dimensional for grey-scale images and 3 dimensional for color images) of pixel values of the images as the features. We can then take this matrix and "flatten" it, meaning that we convert it into a 1-Dimensional array (the form necessary for our intput layer above. Furthermore, we assign to each entry of the resulting vector, i.e. each pixel value a weight, and include it in the calculation of our linear component ("Sum" in the above graph). This means our model is "fully connected". Now that we've defined the functional form of our model and it's architecture, let's discuss how we will train it using our training data set.

Gradient Based Learning

Most machine learning algorithms, including deep learning algorithms, involve minimizing an objective function, $f(x)$, i.e. finding $x^* = \text{argmin} f(x)$. If $f(x)$ is differentiable, one common optimization method involves iteratively taking steps in the feature space such that, in each step, we move in a direction likely to furthest reduce $f(x)$. In other words, we want to find the direction, $\mathbf{u}$ such that the directional derivative $D_{\mathbf{u}} f(x)$ is minimized. (This is an intermediate optimization problem we solve to solve the primary optimization problem of minimizing f.) Then, we take a step of a specifized size, known as the "step size" (which we'll denote by ϵ) in this direction and repeat.

In order to solve this intermediate optimization propblem, recall how partial derivatives are computed. Namely, given a direction $\mathbf{u}$, the directional derivative of $f(x)$ in the direction of $\mathbf{u}$ at a point, $\mathbf{x_0}$, is given by:

$$D_{\mathbf{u}} f(\mathbf{x_0}) = \mathbf{u}^T \nabla f(\mathbf{x_0})$$

where $\nabla f(\mathbf{x_0})$ is the gradient of f at $\mathbf{x_0}$.

For the purposes of our intermediate optimization problem, we can assume (without loss of generality) that u is a unit vector. Thus, $D_{\mathbf{u}} f(\mathbf{x_0})$, can be interpreted as the signed length of the projection of $\nabla f(\mathbf{x_0})$ onto the span of $\mathbf{u}$. Using this interpretation, we can now see that the solution to our intermediate problem is given by $-\nabla f(\mathbf{x_0})$. Using this, at each step in our optimization algorithm, we update our choice of $\mathbf{x}$ as follows:

$$\mathbf{x}' = \mathbf{x} - \epsilon \nabla f(\mathbf{x}).$$

This process is known as "gradient descent", as we are descending down the objective function according to the gradient. While this constitutes the basis for many popular optimization algorithms used in deep learning, it is (by itself) often not the preferred method. In the next section, we will explain, briefly, some of the defects of the algorithm as well as popular alternatives or fixes.

Improving Gradient Descent

Gradient descent uses the gradient of $f(x)$ to determine a step direction. In other words, it uses the first derivatives of $f(x)$, making it a "first-order" optimization method. This presents potential problems, including making choosing an appropriate learning rate very difficult, and an innability to effectively deal with saddle points. To illustrate the former, consider the one dimensional case, i.e. $x \in \mathbb{R}$ and $f(x) \in \mathbb{R}$. According to gradient descent, we move with a fixed learning rate in the direction of $-f'(x)$. If, as we are approaching a local minima, $f''(x)$ is very large, we run the risk of over shooting the minimum with a larger learning rate, while smaller step sizes can cause the algorithm to be inefficient and take many steps to converge.

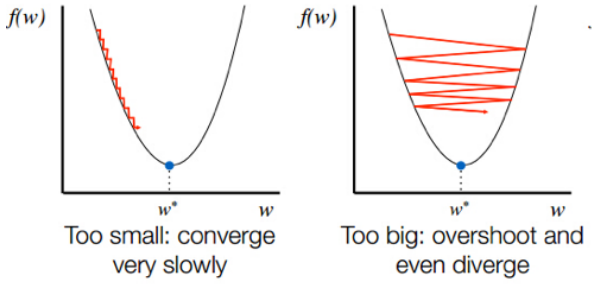
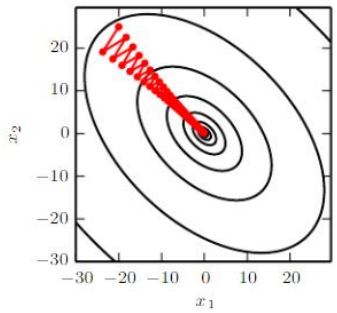


Illustration of poor step sizes in one dimensional gradient descent (Varma & Das, 2018)

This defect persists, analogously, in multiple dimensions. In the image below, we view the trajectory of gradient descent in multiple dimensions using a contour plot representation. Note that the curvature of the objective function is largest in the $[1, 1]^T$ direction and smallest in the $[1, -1]^T$ direction. The objectice function resembles a long canyon (Goodfellow, Bengio & Courville, 2017). Due to a step size which is too large, gradient descent overshoots the bottom of the canyon, having to then descend the opposite canyon wall in the next iteration.



A view of gradient descent on an objective function with ill-conditioned Hessian Matrix (Goodfellow, Bengio, & Courville, 2017)

One approach to addressing these defects is to use information from the second derivative, or Hessian Matrix (the multivariate equivalent) to modify our step direction, making our approach "second-order". Namely, Newton's Method accomlishes this and can sometimes lead to a critical point much faster than gradient descent (especially when the objective function is convex). This is not, however, the approach we focus on here (since most cost functions we'll use are complex and not convex).

Another approach is to smooth out the trajectory of gradient descent by using exponentially weighted averages of gradients, V_{dW}:

$$\begin{aligned} V_{dW} &= 0 && \text{as an initial value.} \\ V_{dW} &= \beta V_{dW} + (1-\beta)dW && \text{for every successive iteration.} \end{aligned}$$

And model parameters are updaed by $\theta = \theta - V_{dW}$. Here β is a parameter of the algorithm to be set. Intuitively, we can think of this algorithm of taking an approximate exponentially weighted average of the previous $\frac{1}{1-\beta}$ terms. While this formulation is intuitive to understand, often, in practice, the update expression is written as $V_{dW} = \alpha V_{dW} - \epsilon dW$ and then parameters are updated by $\theta = \theta + V_{dW}$ (where ϵ need not equal $(1+\alpha)$). This approach is known as gradient descent with "momentum".

Yet another approach, known as Root-Mean-Squared-Propogation (RMSprop), modifies the learning rate in each direction differently and inversely proportional to the square root of an exponentially weighted average of the appropriate directional derivative. Namely:

$$S_{dW} = \beta S_{dW} + (1-\beta)dW^2 \qquad \text{where } dW \text{ is squared elementwise.}$$

And model parameters are updated by:

$$\theta = \theta - \frac{\epsilon}{\sqrt{S_{dW}} + \delta} dW$$

where division and multiplication are performed element-wise and δ is a small constant to stabilize the division operation.

One final approach we discuss is known as ADAM optimization. ADAM optimization combines the momentum and RMSprop algorithms together. At each iterative step, we compute V_{dW} as in gradient descent with momentum (except with a bias correction) and S_{dW} as in RMSprop (except with a bias correction, also). Then updates to model parameters occur as:

$$\theta = \theta - \gamma \frac{V_{dW}}{\sqrt{S_{dW}} + \delta} \text{ for learning rate } \gamma$$

This is the algorithm the authors to train their models, and the algorithm we use to train our logistic regression.

Training and Testing A Logistic Regression Model

In the case of binary classification, it is common to use the binary cross-entropy loss function as $f(x)$. Namely, we want to minimize:

$$f(x) = -\frac{1}{N} \sum_{i=1}^{N} y_i \cdot \log(p(y_i)) + (1 - y_i) \cdot \log(1 - p(y_i))$$

To implement any of the first order algorithms discussed earlier, we need to determine how to compute the gradient of our loss function with respect to the model parameters, $\nabla_\theta J(\theta)$. When information flows forward from the input layer through the model, this is called *forward propogation*. The way we compute ∇_θ is by sending information from the cost back through the network in a process called *back-propagation*. This is as opposed to computing the analytical expression for $\nabla_\theta J(\theta)$ since evaluating this is often computationally expensive as our models get more sophisticated. For the sake of brevity, we will not present the details of back-prop in this report. Instead, we now turn to applying these training methods to our logistic regression model. Namely, we will apply ADAM optimization to our model using the subset of the authors' training data that they provide. The model yields the following evaluation metrics on our test set:

$$\begin{aligned} \text{Accuracy} &: 0.9332 \\ \text{Sensitivity} &: 0.7100 \\ \text{Specificity} &: 0.9407 \\ \text{AUC} &: 0.7665 \end{aligned}$$

As more and more data becomes available to a classification task, the usefullness of deep networks tend to become more and more apparent, as they become more accurate than shallow models (such as our formulation of logistic regression). One such reason for this, is that our logistic regression model is a linear classifier. Namely, it classifies according to a linear decision boundary. Often, if we suspect a non-linear boundary may be more desirable or realistic, we can account for this limitation by applying the linear classifier not to our data directly ($\mathbf{x}$) but to a non-linear transformation of our data ($\phi(\mathbf{x})$). Instead of having to manually engineer a suitable transformation, the approach of deep learning allows the model to learn ϕ. In the next section, we'll briefly introduce ANNs and deep learning as a generalization of sorts of the logistic regression model we've developed.

Introduction to ANNs and Deep Learning

Introduction

Recall from the previous section that Artificial Neural Networks take on the task of learning a suitable non-linear transformation, ϕ to apply to our data. In this spirit, we can think of our network as a composition of functions, $f^{(n)}(\phi(\mathbf{x}))$. The reason we attach the (n) superscript is because, in a deep neural network, we model ϕ itself as a composition of functions: $f^{(n-1)}(f^{(n-2)}(\dots f^{(1)}(\mathbf{x})\dots))$. Thus, in its entirety, a deep neural network can be thought of as a composition of functions: $f^{(n)}(f^{(n-1)}(\dots f^{(1)}(\mathbf{x})\dots))$. Using this framework, $f^{(1)}$ is considered the *first layer* of the network, $f^{(2)}$ is considered the *second layer*, etc. and the length of this chain gives the *depth* of the network. The first layer is often referred to as the *input layer* and the last layer is often referred to as the *output layer*. Layers in between, are often referred to as *hidden layers*. Graphically, we can view these networks as a stacking and composition of multiple logistic regression units (although the sigmoid activation function is often swapped for another -- more on this in a moment):

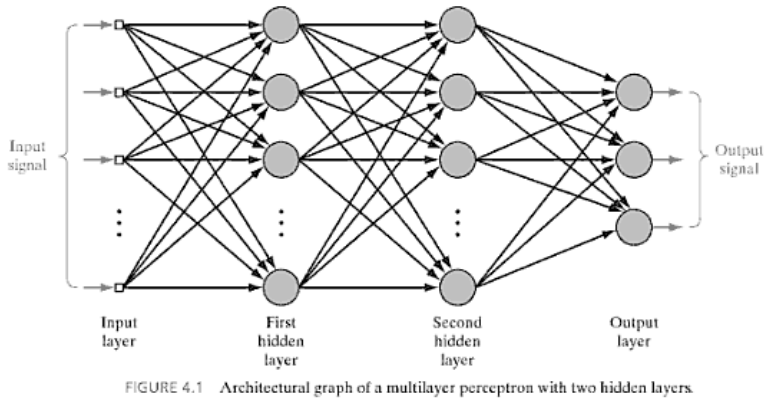


FIGURE 4.1 Architectural graph of a multilayer perceptron with two hidden layers.

MLP graph from (Ogail, 2010).

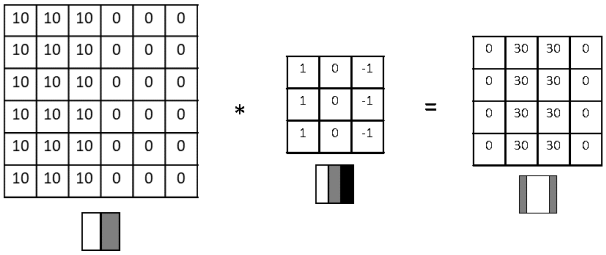
In the figure above, each circle is referred to as a *unit*. Each unit's output value is determined by its inputs as a chosen activation function applied to a linear function of the unit's inputs. The sigmoid function is rarely chosen for input layers. The hyperbolic tangent function is sometimes used, however, the default recommendation for most modern neural networks is the *rectified linear unit* or ReLU function defined by $g(z) = \max\{0, z\}$. Note, we still apply a sigmoid activation at the output layer. More than a slight improvement on the linear classifier of the previous section, the *universal approximation theorems* for neural networks state that a feedforward network with a linear output layer and at least one hidden layer with any appropriate activation functions (sigmoid or ReLU, for example) can approximate any suitably "nice" function from one finite-dimensional space to another with any desired non-zero amount of error, provided that the network is given enough hidden units or layers. ("nice" can mean slightly different things depending on the specific theorem referenced.)

How are deep neural networks trained? The training methods from the previous discussion of our logistic regression model apply equally well to training deep networks. Namely, gradient based methods are popular and utilize back-prop as a means of computing the gradients. Again, for brevity, we omit the details, but the reader can imagine how the methods of the last section generalize, or really, apply directly to deeper networks.

Within the context of computer vision tasks, like the one explored in Minaee et al., there is a popular amendment to the basic fully connected networks we've described till now that has proved useful. This amendment constitutes the inclusion of other types of hidden layers, including what're referred to as convolution layers. Fittingly, these models are referred to as Convolutional Neural Networks, or CNNs. These are the class of models Minaee et al. pull from to use in their paper. In the next section, we introduce some of the basic concepts involved with CNNs.

Introduction to CNNs

Intuitively, we might think that a computer vision task might be made simpler, if we first focus on identifying low level features, like edges, and gradually work up in complexity to things like signs of COVID. While not necessarily a completely accurate description, this conception is a decent heuristic device for thinking about basic convolutional networks. For example, how might we try to force a network to learn about vertical edge detection? One method is to apply a suitable convolution (or sometimes called a filter) to our input arrays of pixels. The *convolution* method of applying such a suitable filter is pictured below:



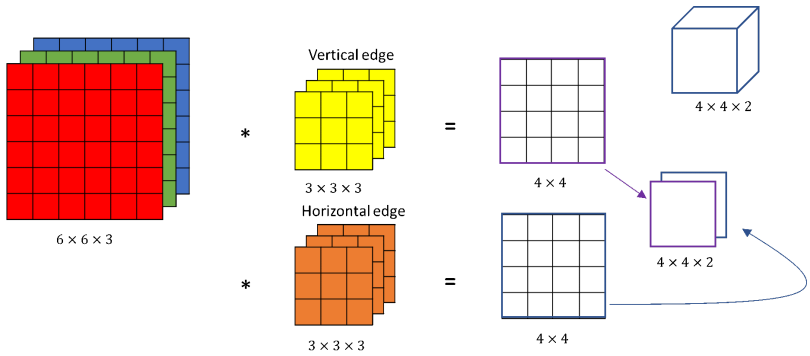
Vertical edge detection through convolution. (Datahacker.rs., 2019)

We see that, for a grey-scale image, the convolution operation --in the CNN sense only; in other areas convolution operations possess different meanings-- consists of overlaying the filter on top of the array, starting at the upper, left hand corner, multiplying the filter values with the pixel values, summing the products together and using the resultant value as the first entry of the output array. We continue in this way, sliding the filter from left to right and top to bottom. It follows, then, that given an input array of size $(n \times n)$ and a filter of size $(f \times f)$, the array resulting from convolution will be of size $(n - f + 1) \times (n - f + 1)$. In practice, instead of hand-picking the filters to use, we usually set the size of the filter and let our optimization algorithm choose the values.

Now, two potential downsides to this procedure include: 1) Every application shrinks the size of your image. And 2) The pixels in the corners of an image and on the edges contribute less to the result than the pixels towards the inner of the image (since they are utilized less often). To address these, we often *pad* the image before convolving. Paddings simply involves adding a number of layers of zeros around the array of pixels. So, if we pad a $(n \times n)$ array with a padding amount, p, the resulting array will be of dimensions: $(n + 2p) \times (n + 2p)$. Thus, after applying a filter of size, $(f \times f)$, we're left with an array of size, $(n + 2p - f + 1) \times (n + 2p - f + 1)$. There are two padding practices most commonly used. Namely, 1) no padding. And, 2) Padding such that the resulting array after convolution has the same dimensions as the input array ($p = \frac{f-1}{2}$, f is usually odd). These are known as *Valid* and *Same* convolutions, respectively.

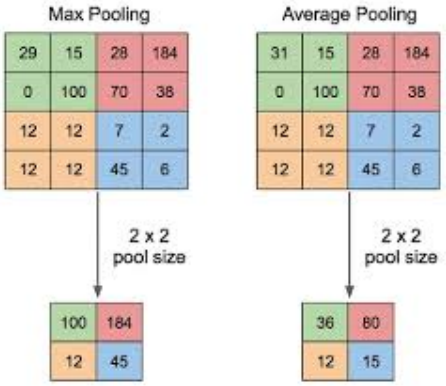
Another common modification to the convolution layer is to skip a set number, s, of columns horizontally and rows vertically every time we shift the filter. This is called a *strided* convolution with stride s and results in an array of size $(\lfloor \frac{n+2p-f}{s} \rfloor + 1) \times (\lfloor \frac{n+2p-f}{s} \rfloor + 1)$. Strides, used in conjunction with padding, allow you to lower the size of the output, while not systematically ignoring the borders of the image.

Now, how does this process work in the case of a 3-channel image input as opposed to a grey-scale image input? Instead of convolving with a 2-dimensional array, we convolve with a three dimensional array. Namely, each channel of the filter is convolved with the corresponding channel of the input and the results of those three convolutions are summed to produce the result, a 2-dimensional array. To apply multiple filters at a time (suppose we want to detect vertical and horizontal edges in the same layer), we can apply each of the 3-dimensional filters to our input and each 2-dimensional intermediate result will constitute a single channel of the final 3-dimensional result:



A 3-dimensional convolution with 3-dimensional output. (Datahackers.rs., 2020)

There is one more aspect usually found in CNNs that we need to introduce and that is the addition of *pooling* layers. Given a filter and stride size for our pooling layer, for a grey-scale image, we consider only the pixels in the upper, left-hand corner of the image of size ($f \times f$). In "Max-Pooling", the first entry of our result will be determined by the maximum of the pixels being considered. Then, the pixels to be considered moves across and down the image, as with convolutions. Another form of pooling occasionally used is "Average-Pooling" where, instead of taking the maximum of the pixels in consideration, we take the average.



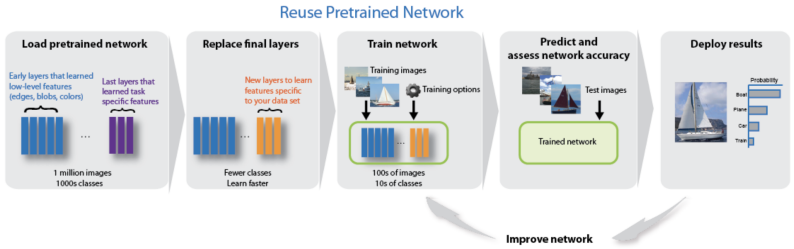
The models used by Minaee et al. in Deep-COVID are special network architectures that use these methods. In

Models Utilized in Minaee Et al.

Given limited number of training data available on COVID at the time of study, Minaee et al. used transfer learning technique to adapt pre-trained deep neural networks for COVID-19 recognition, rather than building the models from scratch.

Transfer learning

Transfer learning essentially is a technique appling "knowledge" from previous tasks to solve "new" yet similar and relative problems. In essense, a base network is trained for a generalizable task, and then the target dataset is used to retrain and fine-tune the model for new classification task. This is a very popular technique in deep learning and in the field of image processing and recognition where enormous sources is required to well train a model. And it's also widely used for COVID diagnosis using chest X-ray, especially at the early stages of pandemic, when little training and testing data is available yet rapic diagnose methods is of great interest and high demand. A set of pre-trained deep learning models trained on ImageNet dataset are publicly available and can be directly incoorporated into new models for reuse.



Transfer learning to retrain models. (Matlab)

Once the pre-trained model is selected, there are two main ways of re-using the model for new target task. One is to use the pre-trained model parameters as initial values,and then update the whole network using training data. Alternatively, one can directly use most of the internal weights in the pre-trained model, and only updating the last learnable layer that's combining the extracted features, as well as the classification layer that's producing class probabilities and predicting labels. This can be realized by "freezing the weights of initial layers" by setting the learning rates from earlier layers in the network to zero. The latter approach is what's used by Minaee et al. in their Deep-COVID study. The pre-trained models were essentially treated as "feature extractor", while the last layers for classifier were replaced with new layers refined for the COVID classification using the training data.

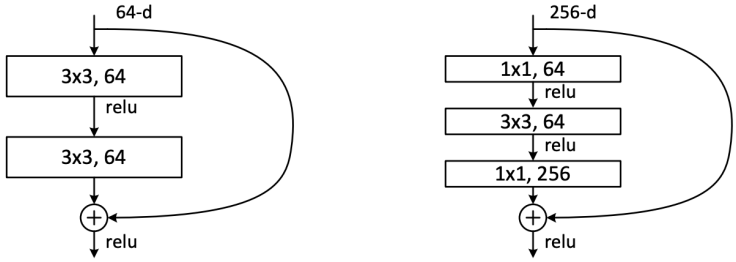
To re-train the models, Minaee et al. used 420 augmented COVID-19 images from COVID-Chestray-Dataset and 2000 non-COVID images from Chex-Pert dataset as introduced above. They fine-tuned each model for 100 epochs, with 20 batches each. ADAM optimzer is used to optimize the cross-entropy loss function, with learning rate being 0.0001.

Pre-trained models utilized in Deep-COVID study

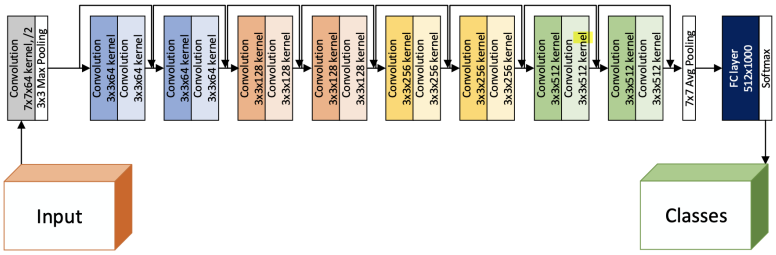
Deep-COVID study tested four popular pre-trained models from essentially 3 classes, namely ResNet (developed by He et al. 2015), SqueezeNet (developed by Iandola et al. 2016), and DensNet (developed by Huang et al. 2017). Required input image for these networks are down-sampled Chest X-ray images of size 224-by-224-by-3, where 224-by-224 indicates the image size, and 3 indicates 3 color channels. All models were originally trained using images from ImageNet (http://www.image-net.org/ (http://www.image-net.org/)).

ResNet

ResNet network was developed by He et al. (He et al. 2015) to address the *degradation* problem of deep CNN, where the training and testing accuracy decreases with the increasing depth of the network. Key innovation of the ResNet network is the deep residual learning. Instead of learning original, unreferenced mapping from x to $H(x)$, the model learns only residual mapping $F(x, \{Wi\}) = H(x) - x$. And it's realized by a *identity shortcut connection* and element-wise addition, where x, or the parameter-free *identity* mapping, is directly forwarded to the next layer via a bypass route. The left figure below demonstrates building block of such structure used for ResNet18. Overall architecture of ResNet50 is very similar to that of ResNet18. The main difference is that the building blocks of ResNet50 used a 3-layer bottleneck design.(As demonstrated in the right figure) It first reduces then restores the dimensions, doubling the model size and increasing the depth of the model. The *identity shortcut connection* helps maintaining the model efficiency, reduces computational burden and minimizes degradation with considerably increased depth.

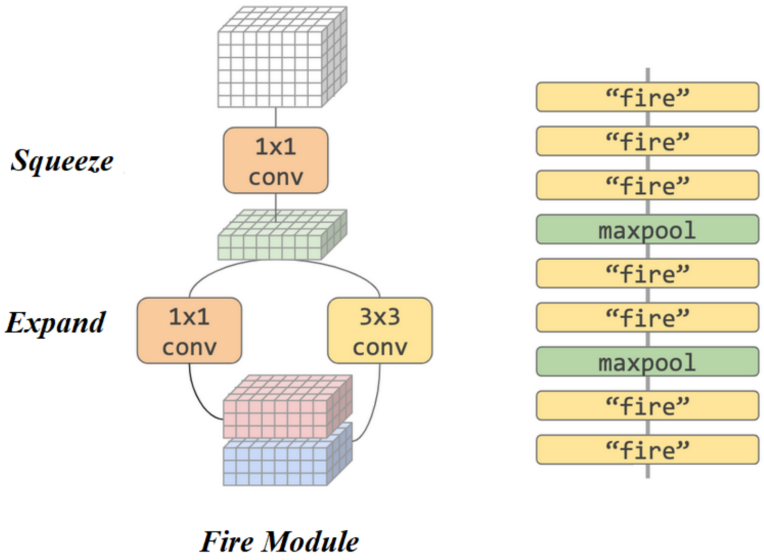


The overall network architecture is demonstrated in the figure below. The base network has 34 weighted parameter layers, with convolutional layers using 3x3 filters, and a global average pooling layer and a 1000-way fully-connected layer with softmax at the end. The pre-trained ResNet models were trained using down-sampled 224x224 images from ImageNet, with decaying learning rate for up to 60×10^4 iterations.



SqueezeNet

Iandola et al. 2016 proposed SqueezeNet network aiming to have a light-weight CNN architecture with fewer parameters that is more efficiently trained and easier to implement, while still achieving same level of accuracy as other CNN networks such as AlexNet. This network is particularly useful in the application of image recognition on automonous driving. Key innovation of the Squeeze network is the use of "fire modules", between the input and final classifier convolution layers starting and ending the network. The fire module replaces the regular 3x3 convolution filters with 1x1 filters, and decreases the number of input channels to 3x3 filters using *squeeze layers*. Then Thus the total quantity of parameters in the layers, calcualted by (number of input channels) x (number of filters x 3x3), are largely decreased. And to still preserve the accuracy, the downsampling was delayed in the network by setting low stride in earlier layers, allowing for larger activation maps for the convolution layers.

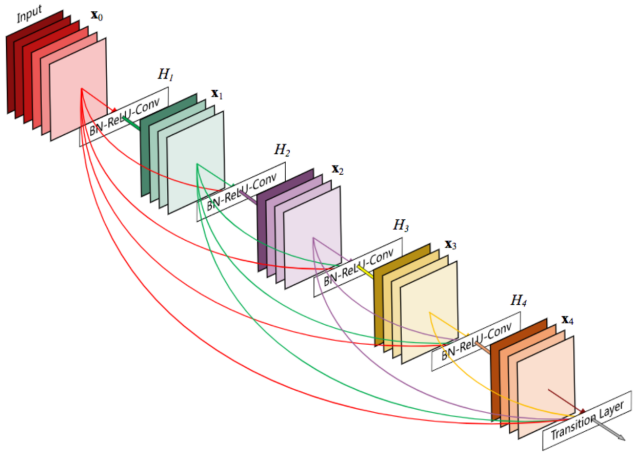


Fire Module

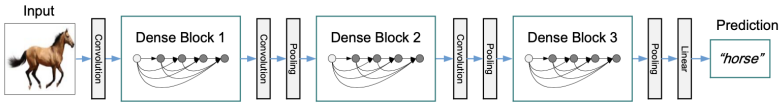
The overall network architecture is demonstrated in the figure below. In SqueezeNet, since the last learnable layer is a 1x1 convolutional layer, to perform the transfer learning, the last convolutional layer need to be replaced with a new convolutional layer with the number of filters equal to the number of classes.(Matlab)

DenseNet-161

DenseNet network was developed by Huang et al. (Huang et al. 2017), to address the vanishing-gradient problem and to reduce the number of parameters at the same time. Unlike Resnet increasing power from increasing depth of the architecture, DenseNet relys on *feature reusing* to draw representational power. Unlike the traditional CNN where each convolution layer is fed and connected only to its neighboring layer, the DenseNet network uses dense block that passes forward the feature-maps to all subsequent layers, thus for each layer in the training process, the features learned in all preceding layers are concatenated and directly further utilized as input by deep layers throughout the same dense block. This improves training efficiency without introducing additional parameters or considerably increasing network depth or width.



The overall architecture is demostrated in the figure. Overall the network is compact, highly parameter efficient, and highly computational efficient. DenseNet is trained on 3 datasets, namely CIFAR, SVHN and ImageNet, using stochastic gradient descent. The pre-trained model used in Deep-COVID study was the DenseNet-161 version of DenseNet.



Evaluation Metrics

Minaee et al. evaluated the model performances on classifying COVID cases using two main metrics: prediction accuracy as measured by sensitivity and specificity of the classification, precision-recall curve. Model-predicted likely regions of COVID-19 was also visually assessed against the radiologist-marked regions, by overlaying model-predicted-region saliency map on top of the Chest X-ray images.

Prediction accuracy

First, the probability score of COVID was calcualted for each test image, incicating the likelihood of the image being detected as COVID-19. Then based on cut-off threshold of selection, the image can be either classified as COVID or non-COVID.

Thus validating the predicted disease status to true image labeling, prediction accuracy thus can be measured by: sensitivity, the percentage of correctly predicted COVID images; and specificity, the percentage of correctly predicted non-COVID images. Confidence interval of the accuracy can be calculated using:

$$r = z_{\alpha/2}\sqrt{\frac{accuracy(1-accuracy}{N}}$$

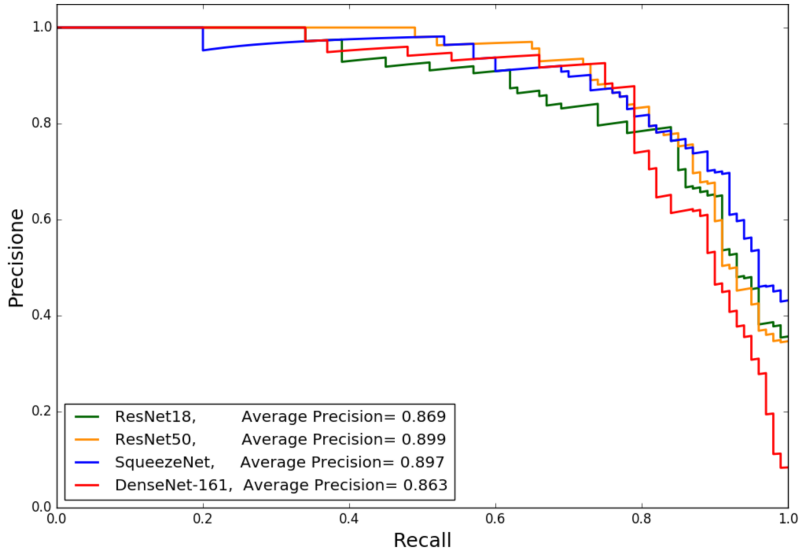
where accuracy is either sensitivity and specificity, N is the total number of images by disease status.

Since the choice of cut-off threshold affects the classification of the predicted COVID-status thus the prediction accuracy, cut-off corresponding to a sensitivity rate of 98% was selected for each model to compare performace across models. Most model, with 98% sensitivity, achieved specificity that's much higher comparing to the classification model using logistic regression shown above. Among the four, Densenet-121 shows lower accuracy comparing to the others.

Model	Sensitivity	Specificity
ResNet18	98% ± 2.7%	90.7% ± 1.1%
ResNet50	98% ± 2.7%	89.6% ± 1.1%
SqueezeNet	98% ± 2.7%	92.9% ± 0.9%
Densenet-121	98% ± 2.7%	75.1% ± 1.5%

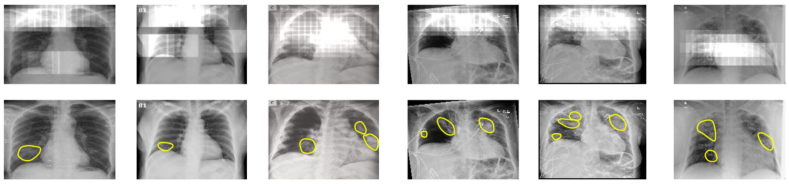
Precision-recall curve

A more comprehensive comparison between models was through the precision-recall curve, where precision rate, the percentage of true positive among all predicted positive, was examined for all levels of sensitivity rate. The average precision was evaluated for quantitative comparison. The figure shows that precision for DenseNet-161 dropped faster as the sensitivity rate reaches 0.8, while precision of SqueezeNet gradually stood out at high sensitivity level. Given the highly imbalanced nature of available data used for analysis, the author considered the precision-recall curve a preferred evaluation metric over the regular Receiver operating characteristics (ROC) curve and AUC.



Model-predicted likely regions of COVID-19

Model-predicted infected region of COVID-19 was defined if occluding that region would cause the model to mis-classify a COVID image as non-COVID. The COVID-19 infected regions detected by the model was overlaid on the Chest X-ray images, and visually assessed against the radiologist-marked regions. The authors claimed overall good aggrement between model-predicted and radiologist-determined regions of COVID infection.



Discussion

In this report, we reviewed theoretical foundations of deep ANN and CNN, as well as the application of CNN in image recognition demonstrated through COVID diagnosis using Chest X-ray images in Mniaee et al. 2020 paper. Overall, deep neural network is a very useful tool for processing big data for complex classification task such as image recognition. It largely increased the accuracy of classification, comparing to linear classifier such as logistic regression models. Though in general deep learning models require large resources to train, transfer learning allows people to fine-tune pre-trained high-performance model for new classification tasks. This can be particularly helpful in a case like COVID, when medical image classification for an emerging disease is in urgent need while lacking adequate images to train a deep learning model from scratch.

In Minaee et al., four popular CNN networks, namely ResNet18, ResNet50, SqueezeNet and DenseNet-161, were re-trained with COVID-19 images from COVID-Chestray-Dataset and non-COVID images from Chex-Pert dataset. Most models had satisfying performance in classifying COVID images, with average precision about 0.86~0.90, and model-predicted infection region in overall good agreement with radiologist-identified regions for COVID. The main limitation of the Deep-COVID study, as pointed out by the author, is limited size of training and testing data. Future research can consider using larger training and testing datasets for more reliable estimation of prediction accuracy, for example, the nearly released open access benchmark dataset, COVIDx, with nearly 14 thousands COVID Chest X-rays.(Wang et al. 2020)

Publications on utilizing deep learning and CNN for COVID diagnosis using Chest X-ray images has been exploding in the past few months. The Deep-COVID study was among the first batches of publications, thus the authors didn't include rationals of choosing these four pre-trained model over the other models, nor did they systematically review or compare across publications using other models for the same task. In fact, a wide variety of pre-trained models has been adapted for this task, even a few new models have been developed, yet little has been done to systematically compare the performances of different CNN networks on COVID diagnosis, or to explore how fine-tuning existing CNN is comparing to new models designed and trained for COVID diagnosis tasks, and redundant effort has been spotted. One recent publication, Wang et al. 2020, developed a new CNN network, COVIDnet, trained and tested using COVIDx. The results show that COVID-Net out-performed ResNet-50 in terms of sensitivity of prediction. However, such comparison is limited given current large body of literatures and might not be so useful in the practical sense to inform which architecture should be used to assist COVID diagnosis in health facilities. A systematic review of existing literatures could be useful.

Architecture	Params (M)	MACs (G)	Acc. (%)
VGG-19	20.37	89.63	83.0
ResNet-50	24.97	17.75	90.6
COVID-Net	**11.75**	**7.50**	**93.3**

Table 1. Performance of tested deep neural network architectures on COVIDx test dataset. Best results highlighted in bold.

Architecture	Normal	Non-COVID19	COVID-19
Sensitivity (%)			
VGG-19	**98.0**	90.0	58.7
ResNet-50	97.0	92.0	83.0
COVID-Net	95.0	**94.0**	**91.0**

Table 2. Sensitivity for each infection type. Best results highlighted in bold.

One other limitation of Deep-COVID study is the capability of the models in distinguishing COVID-19 from other respiratory diseases that might also have signs on Chest X-rays, such as non-COVID-19 pneumonia. The training and testing datasets for non-COVID images used in the Deep-COVID study were a mixture of no-finding and 13 categories of non-COVID diseases with or without potential X-ray symptoms. Thus the specificity

reported in the article might potentially be overestimated, since it's reasonable to suspect that classifying COVID-19 images and non-COVID-pneumonia images would be a harder task than classifying COVID-19 images from non-pneumonia images. Some other study, such as Zhang et al. 2020, tested model performance of CV19-Net on classifying COVID-19 pneumonia and non-COVID-19 pneumonia using Chest X-rays all from pneumonia patients, and this approach could be incoorporated into future research for more valid estimation of prediction accuracy.

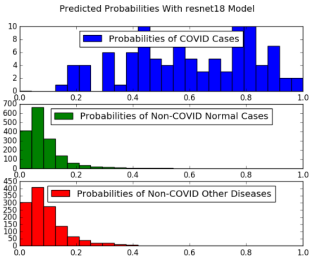


Fig. 6. The predicted probability scores by ResNet18 on the test set.

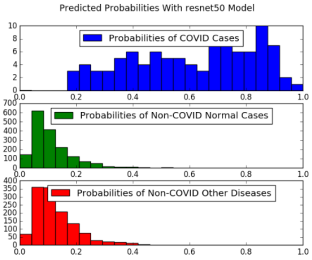


Fig. 7. The predicted probability scores by ResNet50 on the test set.

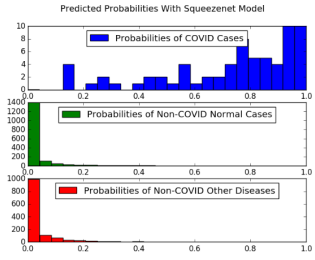


Fig. 8. The predicted probability scores by SqueezeNet on the test set.

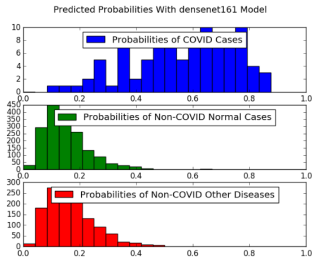


Fig. 9. The predicted probability scores by DesneNet-121 on the test set.

Bibliography

1) Minaee, Shervin et al. “Deep-COVID: Predicting COVID-19 from chest X-ray images using deep transfer learning.” Medical image analysis vol. 65 (2020): 101794. doi:10.1016/j.media.2020.101794

2) Goodfellow, I., Bengio, Y., & Courville, A. (2017). Deep learning. Cambridge, MA: MIT Press.

3) Varma, S., & Das, S. (2018, September 27). Deep Learning. Retrieved November 24, 2020, from https://srdas.github.io/DLBook/ (https://srdas.github.io/DLBook/)

4) Deshpande, M. (2020, September 29). Perceptrons: The First Neural Networks. Retrieved November 23, 2020, from https://pythonmachinelearning.pro/perceptrons-the-first-neural-networks/ (https://pythonmachinelearning.pro/perceptrons-the-first-neural-networks/)

5) Ogail, A. (2010, December 27). Multilayer Perceptron. Retrieved November 25, 2020, from https://elogeel.wordpress.com/2010/05/10/multilayer-perceptron-2/ (https://elogeel.wordpress.com/2010/05/10/multilayer-perceptron-2/)

6) Datahacker.rs. (2019, October 20). #003 CNN More On Edge Detection. Retrieved November 25, 2020, from http://datahacker.rs/edge-detection-extended/ (http://datahacker.rs/edge-detection-extended/)

7) Datahacker.rs. (2020, April 28). #006 CNN Convolution On RGB Images. Retrieved November 25, 2020, from http://datahacker.rs/convolution-rgb-image/ (http://datahacker.rs/convolution-rgb-image/)

8) Matlab. Deep Learning Toolbox. Retrieved December 01,2020, from https://www.mathworks.com/help/deeplearning/ug/train-deep-learning-network-to-classify-new-images.html (https://www.mathworks.com/help/deeplearning/ug/train-deep-learning-network-to-classify-new-images.html)

9) He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

10) Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., & Keutzer, K. (2016). SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. arXiv preprint arXiv:1602.07360.

11) Huang, G., Liu, Z., Van Der Maaten, L., & Weinberger, K. Q. (2017). Densely connected convolutional networks. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 4700-4708).

12) Wang, L., Lin, Z. Q., & Wong, A. (2020). Covid-net: A tailored deep convolutional neural network design for detection of covid-19 cases from chest x-ray images. Scientific Reports, 10(1), 1-12.

13) Zhang, R., Tie, X., Qi, Z., Bevins, N. B., Zhang, C., Griner, D., ... & Li, K. (2020). Diagnosis of covid-19 pneumonia using chest radiography: Value of artificial intelligence. Radiology, 202944.