Pneumonia X-ray Convolutional Neural Network

Oliver Vazquez, Jeffrey Zhu

Project Description:

The purpose of this project is to implement a Convolutional Neural Network (CNN) to identify the presence of bacterial and viral pneumonia in a chest x-ray.

Definitions:

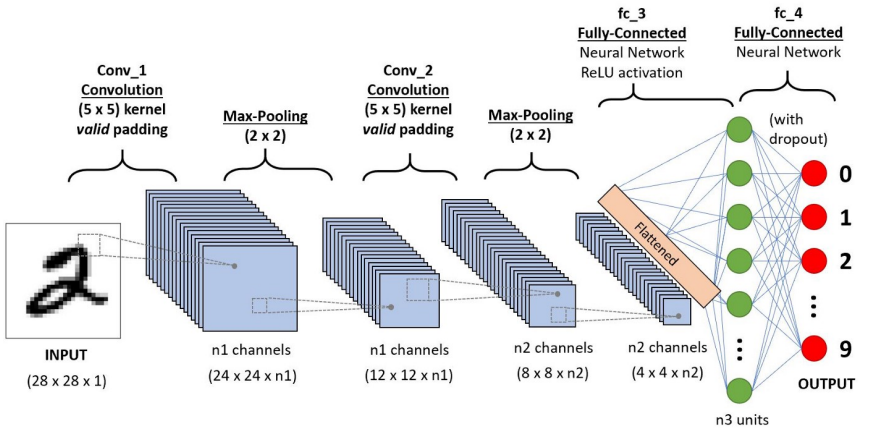
Neural Network:

A Neural Network is a method of deep learning that can translate a series of inputs into a desired output. It accomplishes this through training data that have both the inputs and desired outputs provided. As it's name may suggest, a machine learning neural network operates in a similar way as the neurons in our brains. It strengthens and weakens the bond between each node in the network to handle inputs accordingly

https://deepai.org/machine-learning-glossary-and-terms/neural-network

Convolutional Neural Network (CNN):

A Convolutional Neural Network is a specialized version of a traditional neural network. This type of neural network is typically used with image inputs. Below is a visual depiction of each step required to convert an image into a desired output.

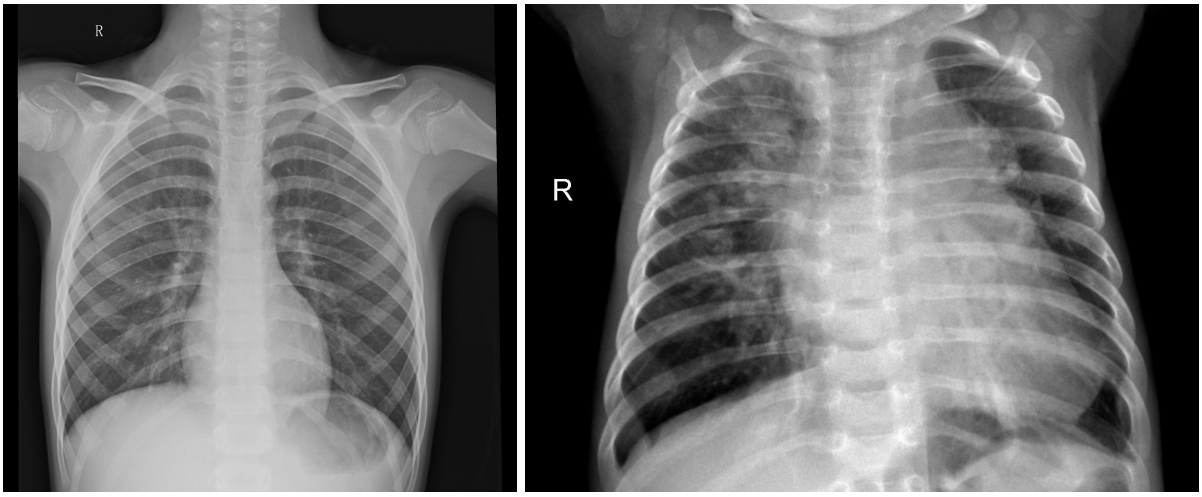


https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

Pneumonia:

Pneumonia is typically classified as an air sac infection that causes inflammation of the lungs. This inflammation causes pus or fluids to fill the air sacs. This, along with the inflammation causes white spots on the lungs which can be seen in a chest x-ray. Below are two chest x-ray images from the dataset. The left chest x-ray is normal, and the right is of pneumonia.

https://www.mayoclinic.org/diseases-conditions/pneumonia/symptoms-causes/syc-20354204



Motivation:

When picking a topic, we were very interested in completing a project that involves neural networks. Convolutional neural networks happen to be both very well supported by tensorflow and keras, but also have a very tangible and easily verifiable input and output. Once we decided to complete a CNN related project, we began to browse Kaggle (an online database for datasets). The dataset “Chest X-Ray Images (Pneumonia)” compiled by Paul Mooney was a gold rated dataset with a large amount of upvotes, and also had a lot of activity and code submissions. We took this as a good sign that it was a solid dataset and moved forward with it. However, we found a comment by a user named Tolga Dincer who raised the important point that the training and testing sets were split in a poor way. The original dataset was split into 3 categories, training, testing, and val. The val dataset however only had 16 images to be used to validate a training dataset of 1000 images. He provided a link to a version of the dataset that contained the same images, however it is split evenly into two categories, training and testing. This is the dataset we ended up using.

Inputs/Outputs:

As we previously discussed, the inputs to the neural network are chest x-ray images, and the outputs will be normal or pneumonia. Ideally, external images completely unrelated to this database will be able to be used as validation. This would ensure that our neural network is not overfitting for this specific dataset, as well as prove that our model is more useful than a mere novel implementation.

Dataset:

The dataset we chose contains two folders, test and train. The test folder contains 234 normal images, and 390 pneumonia images. The train folder contains 1349 normal images, and 3883 pneumonia images. The pneumonia images are further split into bacterial and viral pneumonia. We are currently not classifying the pneumonia images into their specific type of image. Additionally, the type of pneumonia resulting from Covid-19 appears differently from that of viral or bacterial pneumonia. A possible future adaptation of this neural network could include classifying the type of pneumonia.

Methods/Algorithms:

The source code for this project can be found in the following repository:

https://github.com/olivervz/Pneumonia-Neural-Network

Instructions are provided in the README.md to very easily setup an environment and complete all of the below steps. This has only been tested on linux, but only requires python to operate.

Our project has three steps. The first step is to convert the .JPEG images into a usable numpy array that can be interpreted by our model. This is done with the following function.

main/load.py 37

```
# A user submitted a function for the same kaggle dataset which converts the
# images into numpy arrays that can be used by Keras.
# https://www.kaggle.com/madz2000/pneumonia-detection-using-cnn-92-6-accuracy
def load_dataset(dir):

    dataset = []
    for xray_type in xray_types:
        # Get path to the x-ray image
        path_to_images = os.path.join(dir, xray_type)
        # 0 for NORMAL, 1 for PNEUMONIA
        class_num = xray_types.index(xray_type)
        # List all images in each directory
        for xray_image in os.listdir(path_to_images):
            # Get path to each image
            path_to_image = os.path.join(path_to_images, xray_image)
            # Use the cv2 library to convert the GRAYSCALE image into an array
            img_arr = cv2.imread(path_to_image, cv2.IMREAD_GRAYSCALE)
            # Resize the array
            resized_arr = cv2.resize(img_arr, (img_size, img_size))
            # Append each image array with it's type
            dataset.append([resized_arr, class_num])
    # Return the array as a numpy array
    return np.array(dataset, dtype=object)
```

The input to this function is the path to a directory containing two folders, NORMAL, and PNEUMONIA. The output of this function is a numpy array of arrays, where each array contains two elements, an array representing the image, and a 0 or 1 indicating NORMAL or PNEUMONIA. The function operates by using the open cv2 function imread(). imread() accepts a path to an image, and an optional flag indicating what colorscheme the image should be read in, and returns a numpy.ndarray. A numpy.ndarray is a multidimensional array of items of the same type and size, in this case a 2D array where each value represents the specific pixel's value. Next, each image array is resized to 250x250 using the cv2 function resize().

Since this process takes a while, the datasets are saved to .npy binary files using np.save().

The next step is to train the data. First, the data is loaded from the .npy binary files using np.load(). Next, the data is normalized. Prior to this step, each element of the image array has a value between 0 and 255. This value indicates the pixel's grayscale value, 0 being black, and 255 being white. In order for this to be a valid input to our neural network, each value needs to be a float value between 0 and 1, this is done by dividing each element by 255.

Next, the model is constructed. To construct this model, we used a fairly basic CNN structure with one layer for each stage.

main/train.py 52

```
def generate_model():
    model = Sequential()
    model.add(Conv2D(28, kernel_size=(3, 3), input_shape=(img_size, img_size, 1)))
    model.add(MaxPool2D(pool_size=(2, 2)))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(10, activation='softmax'))
    model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
    return model
```

Sequential() allows us to build our model layer-by-layer.

Conv2D() creates a 2D convolution layer. In this layer, the kernel (also known as the filter) is a matrix that is used to extract the features from an image. The kernel "slides" over the height and width of the image and performs the dot product with a subregion, summing up the results into a single output pixel. The kernel performs this same operation for every subregion it slides over, and the output is a matrix of dot products also known as an activation map (or feature map). The activation map is simply a different 2D matrix representation of the image and its high-level features.

Next, max pooling is done via MaxPool2D() to downsample the activation map by summarizing the presence of features in patches (defined by the pool_size parameter) of the activation map. Max pooling, as the name implies, accomplishes this by taking the maximum value over the patch for each dimension along the activation map. Overall, this pooling layer reduces the spatial size of the representation and decreases the computational cost.

Flatten() is used to flatten the input. In other words, the matrix is now converted into a single 1D array.

In the dense layer created by Dense(), we define the output size of this layer and set an

element-wise activation function to be used for the output values. In our model, we use the relu function and softmax function. Relu applies the rectified linear activation function, which returns max(x,0), the element-wise maximum of 0 and the input unit. Softmax converts a real vector into a vector of categorical probabilities. We use this activation function in the last layer of our model so that the result could be interpreted as a probability distribution.

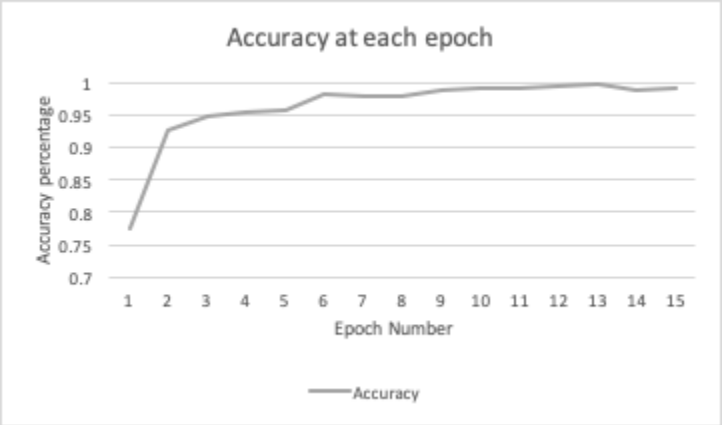
Dropout() helps prevent the model from overfitting. Overfitting occurs when the model learns from the training data “too well.” When a model conforms too closely to the training data, a significant amount of its predictive power is lost. To combat this, our dropout layer randomly sets inputs to 0 with a frequency of 0.2 at each step during training. Inputs not set to 0 are scaled up by 1/(1-0.2)=1.25 such that the sum over all inputs remain unchanged.

Compile() defines the loss function, optimizer, and metrics. In our model, we use the Adam algorithm for our optimizer and sparse categorical cross-entropy for our loss function. We decided to use sparse categorical cross-entropy because our classes (NORMAL, PNEUMONIA) are mutually exclusive, and thus an image can belong to only one of these categories.

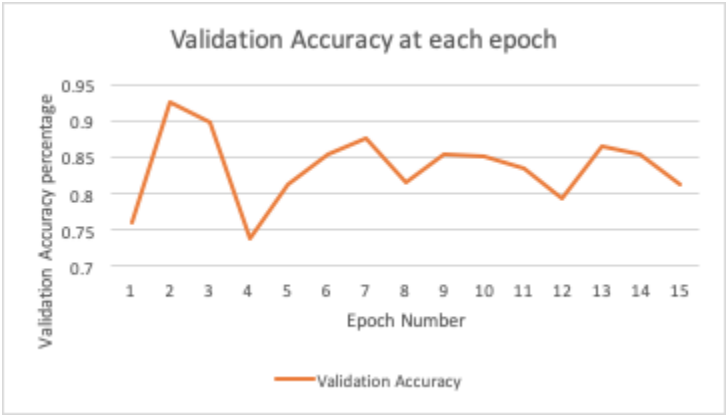
Finally, this model is trained using Model.fit(). This builds our model over the specified number of epochs. In keras, an epoch is defined as one pass over the entire dataset. We arbitrarily set our number of epochs to 15 to see when convergence would occur. Each epoch generates a model accuracy, and loss, as well as the time required for that epoch. A screenshot of the training is provided below.

```
_________________________________________________________________
2021-04-28 21:24:28.042094: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:116] None of the MLIR optimization passes are enabled (registered 2)
Epoch 1/15
164/164 [==============================] - 86s 523ms/step - loss: 5.7706 - accuracy: 0.7734 - val_loss: 0.6459 - val_accuracy: 0.7590
Epoch 2/15
164/164 [==============================] - 88s 540ms/step - loss: 0.1712 - accuracy: 0.9276 - val_loss: 0.1802 - val_accuracy: 0.9251
Epoch 3/15
164/164 [==============================] - 103s 630ms/step - loss: 0.1425 - accuracy: 0.9489 - val_loss: 0.2562 - val_accuracy: 0.8974
Epoch 4/15
164/164 [==============================] - 94s 574ms/step - loss: 0.1327 - accuracy: 0.9534 - val_loss: 1.1063 - val_accuracy: 0.7378
Epoch 5/15
164/164 [==============================] - 96s 583ms/step - loss: 0.1106 - accuracy: 0.9573 - val_loss: 0.6139 - val_accuracy: 0.8127
Epoch 6/15
164/164 [==============================] - 96s 583ms/step - loss: 0.0611 - accuracy: 0.9807 - val_loss: 0.4294 - val_accuracy: 0.8550
Epoch 7/15
164/164 [==============================] - 96s 585ms/step - loss: 0.0623 - accuracy: 0.9775 - val_loss: 0.4144 - val_accuracy: 0.8762
Epoch 8/15
164/164 [==============================] - 95s 579ms/step - loss: 0.0571 - accuracy: 0.9802 - val_loss: 0.6725 - val_accuracy: 0.8160
Epoch 9/15
164/164 [==============================] - 94s 574ms/step - loss: 0.0389 - accuracy: 0.9870 - val_loss: 0.5062 - val_accuracy: 0.8534
Epoch 10/15
164/164 [==============================] - 98s 595ms/step - loss: 0.0287 - accuracy: 0.9917 - val_loss: 0.6156 - val_accuracy: 0.8502
Epoch 11/15
164/164 [==============================] - 127s 773ms/step - loss: 0.0289 - accuracy: 0.9898 - val_loss: 0.7283 - val_accuracy: 0.8355
Epoch 12/15
164/164 [==============================] - 112s 682ms/step - loss: 0.0224 - accuracy: 0.9928 - val_loss: 0.9985 - val_accuracy: 0.7932
Epoch 13/15
164/164 [==============================] - 107s 655ms/step - loss: 0.0143 - accuracy: 0.9957 - val_loss: 0.6021 - val_accuracy: 0.8648
Epoch 14/15
164/164 [==============================] - 109s 662ms/step - loss: 0.0284 - accuracy: 0.9875 - val_loss: 0.5966 - val_accuracy: 0.8534
Epoch 15/15
164/164 [==============================] - 107s 652ms/step - loss: 0.0212 - accuracy: 0.9922 - val_loss: 1.0766 - val_accuracy: 0.8127
20/20 [==============================] - 4s 176ms/step - loss: 1.0766 - accuracy: 0.8127
Model Loss:  1.0765612125396729
20/20 [==============================] - 4s 182ms/step - loss: 1.0766 - accuracy: 0.8127
Model Accuracy:  81.27036094665527 %
```

The model with the basic layers above results in an accuracy of 81.27%. Each epoch’s loss and accuracy is displayed in the plot below.



What's interesting, is that although our accuracy steadily increases, and more epochs of training are provided, the validation accuracy doesn't increase. The loss seems to plateau at around epoch 10. The accuracy seems to plateau at around epoch 9. These are good indications of when the model has converged. The below plot shows the validation accuracy after each epoch

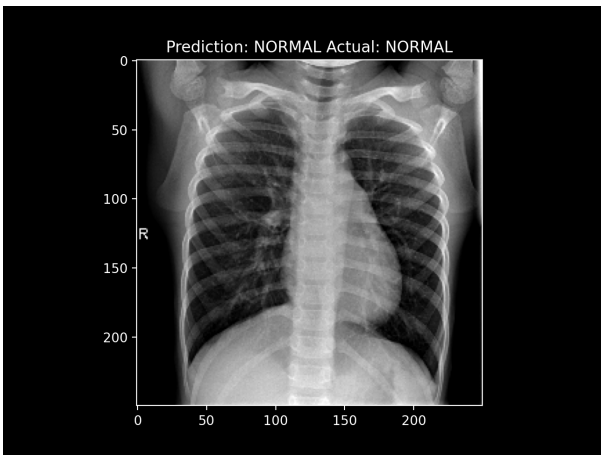


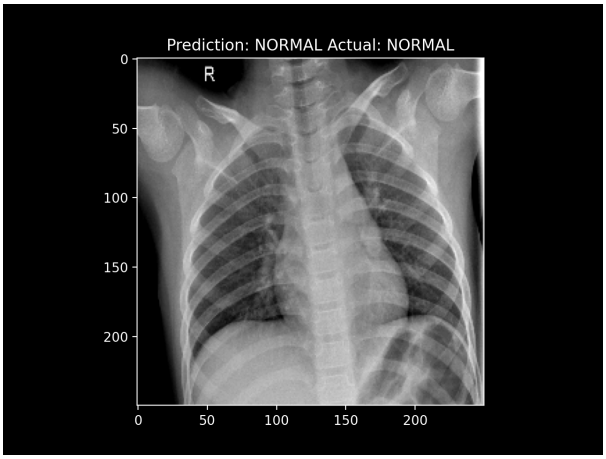
The validation accuracy peaks after 2 epochs at 92.5%, which is much higher than 81.27% after 15 epochs. This may be a result of overtraining. Also, the validation accuracy does not seem to have any relation with our epoch number. This indicates that repeatedly training our model

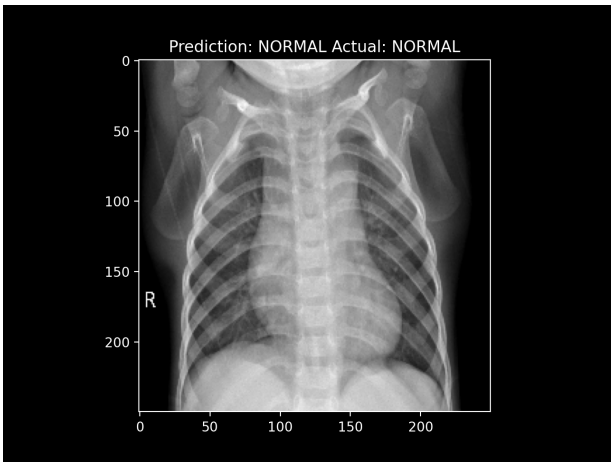
using the same data does not have a significant effect on our model's accuracy.

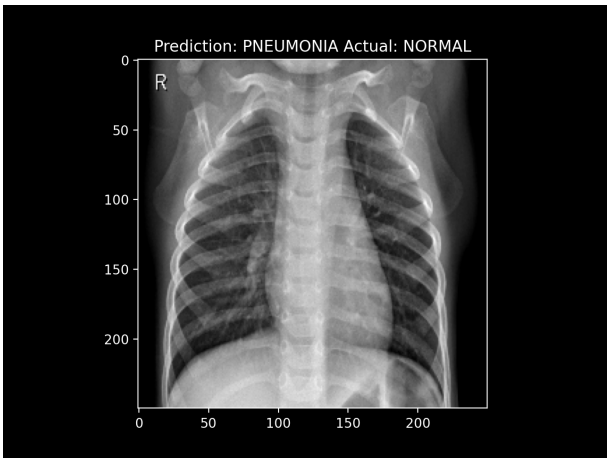
Next, this model's structure is saved to a json file using the Model.to_json() function, and the weights of each node are saved to a .h5 file using the Model.save_weights() function. The model's json structure and the binary file containing its weights can be found in the model/ directory.

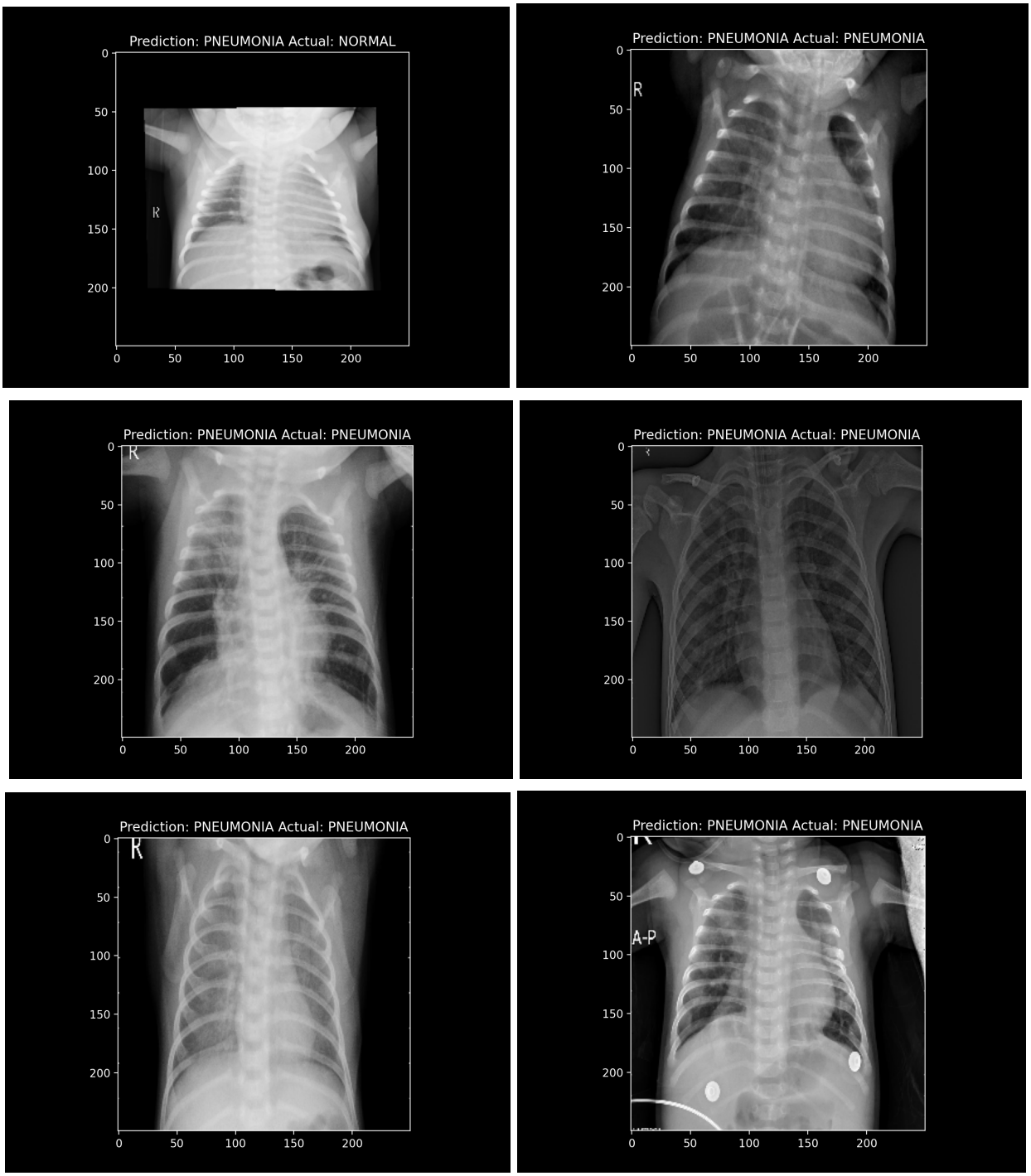
The final step is to run validation with the model. To accomplish this, we extracted 5 pneumonia images, and 5 normal images from the test data. We then used these independant images to test our model. For each image we displayed the image, the model's prediction, and the actual classification. With our above model, and 2 epochs, we correctly identify 8 out of the 10 images, including all 5 Pneumonia images. The output of this validation stage is shown below.





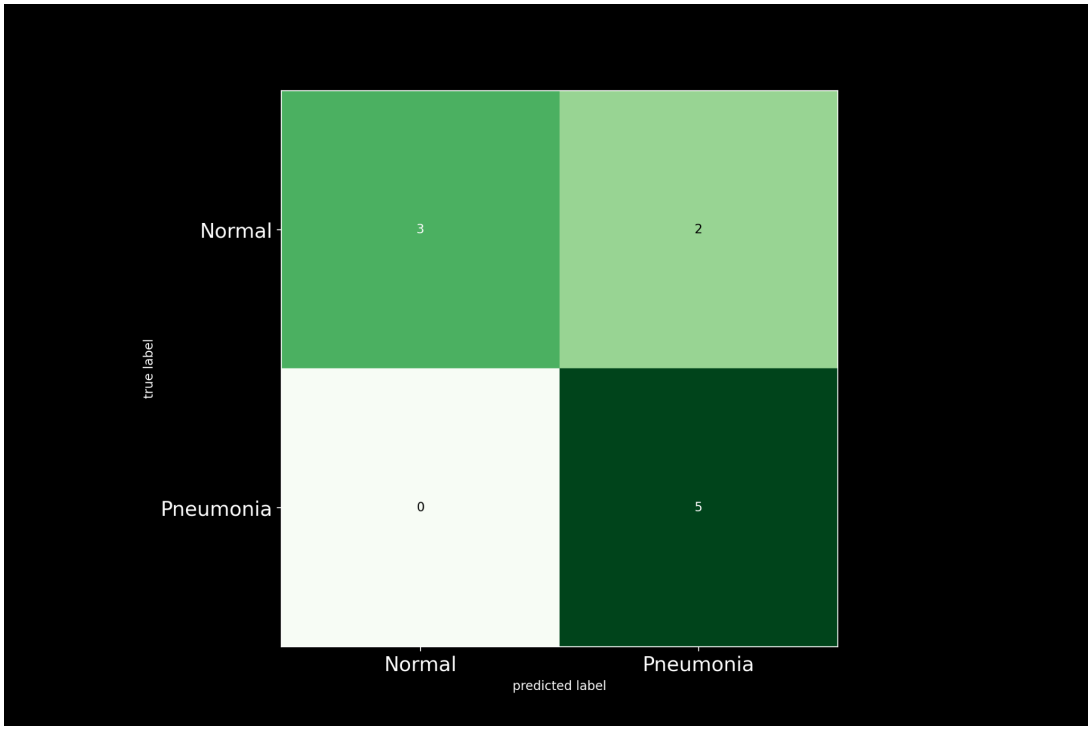




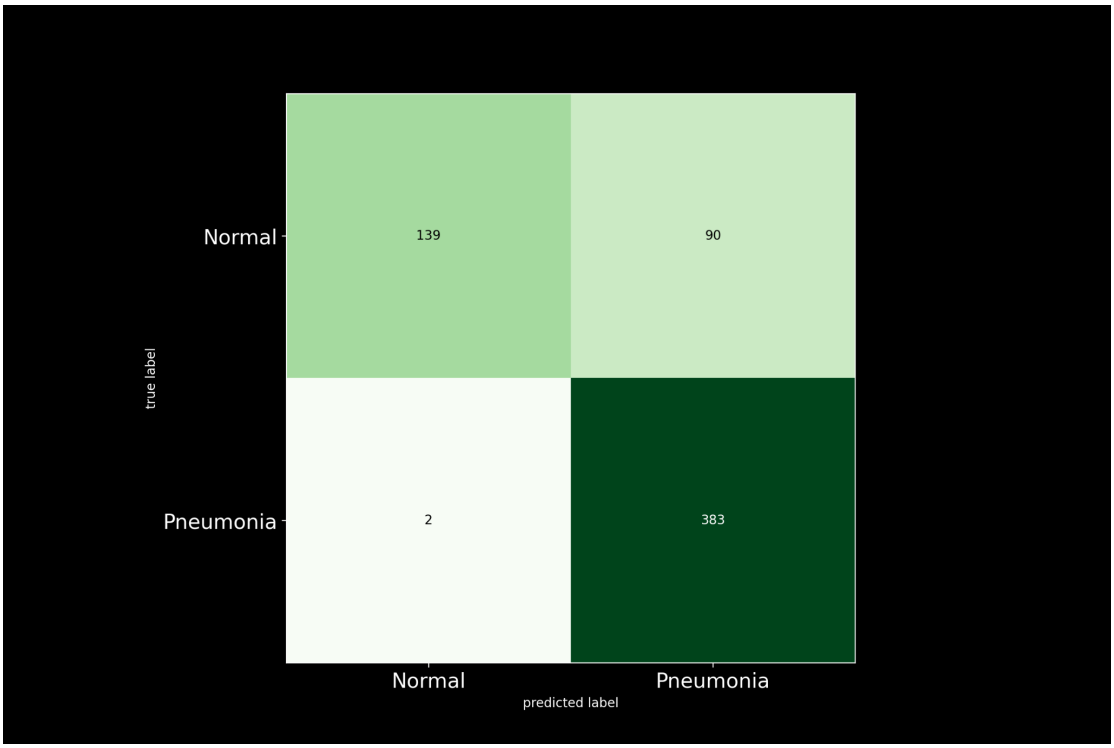


Validation Analysis:

To represent our model's predictions, we created a confusion matrix using mlxtend, a machine learning extension library for python. This generated the following plot.



Our model tends to predict that the x-ray is pneumonia more often than normal. This is most likely due to the fact that there are many more pneumonia images offered by the dataset than normal. Obviously, this is far too few images to generate a consensus, so we repeated this step with the much larger test dataset. This resulted in the similar confusion matrix



The rates for our model's accuracy are as follows (Positive being presence of pneumonia):
True Positive: 383 out of 385, **99.48%**
True Negative: 139 out of 229, **60.70%**
False Positive: 2 out of 385, **0.52%**
False Negative: 90 out of 229, **39.30%**

Our model is very good at identifying the presence of pneumonia in an pneumonia x-ray. It very rarely provides a false negative. A false negative reading is much worse than a false positive. However, our model is much weaker at identifying a normal x-ray as such. Around 40% of the time, our model will misidentify a normal x-ray as one with the presence of pneumonia. A false positive is much better than a false negative, but this is something that needs to be improved

Code References:

Group member Oliver Vazquez had previously completed a project involving the creation of a CNN to identify handwritten digits. This project used the built in MNIST database, but provided a solid base for creating, saving, and loading a CNN model. However, that project involved a build in dataset which lowered the complexity of the project by quite a bit. A Kaggle user named madz2000 submitted their implementation of a CNN (https://www.kaggle.com/madz2000/pneumonia-detection-using-cnn-92-6-accuracy). This was a useful resource, as neither of us had very much experience using cv2 and numpy to convert the .JPEG images in the dataset into usable inputs to the model. However, we will be completing our own analysis and interpretation of the results, as well as creating our own model out of different layers provided by keras.

Discussion/Future Objectives:

One of the difficulties we encountered in completing this project was just getting started in general. We underestimated the amount of time researching and learning about CNNs would take. Also, neither of us have ever really used any of the above math and machine learning libraries before, so that contributed to our list of challenges as well. We initially planned to implement multiple classification algorithms to compare the results of each of them, but scrapped that idea once we realized how much time we needed to devote to learning about CNNs first.

Future work would involve using different model architectures. Our model is very basic, which provided a good starting point, however, using more layers would make our model more accurate.

Additionally, our dataset was skewed towards pneumonia images, this imbalance may cause inaccurate training. This could be offset by adding additional images to the normal x-ray set, or through data augmentation. Something we saw in a lot of CNN examples, was the concept of data augmentation. This involves taking images from a dataset, and altering them to provide a new image to train the model with. These alterations may include rotating the image, zooming in on the image, padding the image with a border, panning the image so some of it is cut off, or

altering the brightness/darkness of an image.

Another future consideration involves finding images completely isolated from our dataset to test our model with. Ideally, our model would be able accurately identify the presence of pneumonia in an x-ray that is visually very different from ones provided in our dataset.