Image Classifier

Ananya Sharma
Department of Computer Science, University of Southern California,
Los Angeles, California 90089, USA

Abstract:

In this report I first describe the kinds of data we have, how to ensure it's all in compliance and can gel along with each other. Then I start by testing a model. Then by measuring its performance metrics namely - Precision, recall, FScore, Accuracy. Then, I compare this model against 2 other models with different activation functions and layers. Thus, finally deciding which model is better by fine tuning the parameters and eventually testing a new dataset to predict labels.

Introduction :

For this task, images of people's lungs have been given to be classified as
0- healthy
1- pre-existing condition
2- mass/effusion

By training my model, these datasets just need to be run on it to get predictions.

Model Selection and Evaluation:

I started by importing the libraries required and the data (a numpy array containing images and a csv file storing its corresponding labels)

```
In [1]: #import libraries and modules
        import numpy as np
        import pandas as pd
        from numpy import genfromtxt

        #load data - images & labels

        data = np.load('/Users/ananyasharma/Downloads/ps4_trainvalid_images.npy')
        a = np.genfromtxt('/Users/ananyasharma/Downloads/ps4_trainvalid_labels.csv', delimiter=",",
```

FIG 1 : How to load data

Then, to know what kind of data we have, its dimensions etc.

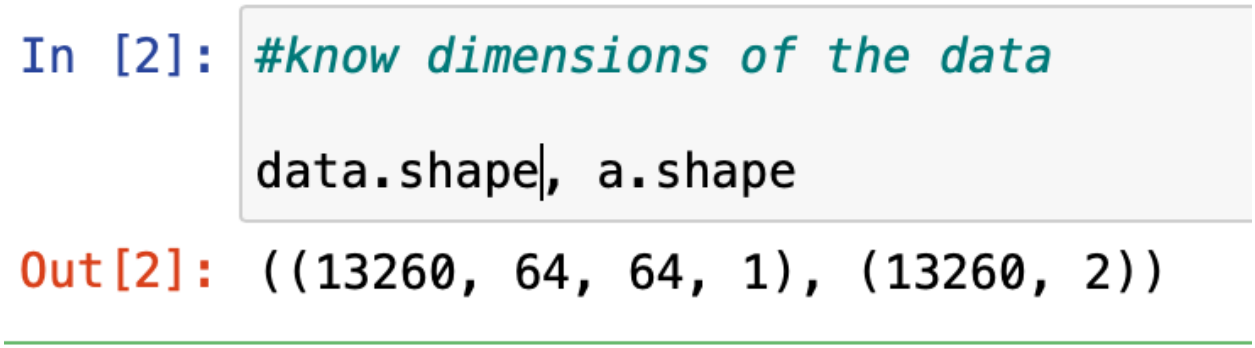


FIG 2 : What dimensions of data

To proceed further, we need to encode the label data. This encoding is known as One Hot Encoding where one hot encoding is a representation of categorical variables as binary vectors.
This first requires that the categorical values be mapped to integer values. Then, each integer value is represented as a binary vector that is all zero values except the index of the integer, which is marked with a 1. SInce we already have the labels in integer value, all we need to do is represent its as binary vectors. This is required due to categorical entropy. Direct integer values couldve been used if there was Sparse categorical crossentropy, however, not all models do and hence encoding is preferred. sparse categorical crossentropy is used when the classes are mutually exclusive (i.e. when each sample belongs exactly to one class) and categorical crossentropy is used when one sample can have multiple classes or labels are soft probabilities (eg - [0.5, 0.3, 0.2]).

Looking at ‘a’ right now we see that it has 2 columns - one for index and one for storing class labels. So, dropping the first column and then encoding will be done.

```
In [5]: a = np.delete(a, 0, axis=1)

In [6]: a

Out[6]: array([[0.],
               [0.],
               [0.],
               ...,
               [0.],
               [2.],
               [0.]])

In [7]: a.shape

Out[7]: (13260, 1)

In [8]: from keras.utils import to_categorical

        a = to_categorical(a)

In [9]: a.shape

Out[9]: (13260, 3)
```

FIG 3 : One Hot encoding of the labels

Now, creating a random index to cross reference the images and its corresponding labels. However, we see that the data is highly unbalanced, and to ensure random splitting of data into training and validation sets, the data will be shuffled as well.

```
In [11]: from sklearn.utils import shuffle

         idx = list(range(len(a)))
         np.random.shuffle(idx)

         idx

Out[11]: [4763,
          2845,
          7267,
          7585,
          4519,
          8860,
          10211,
          7337,
          4650,
          6691
```

FIG 4 : Index created to randomly shuffle the labels and images while keeping their correlation

Now, to proceed further with splitting the data, an index of the length of the number of labels will be used to split the data into 80% training data and the remaining 20% data to be testing data.

```
In [6]: #Split the indices using this index

        N = len(idx)
        train_indices = idx[:int(N*0.8)]
        valid_indices = idx[int(N*0.8):]
```

```
In [7]: #Splitting data into training and validation sets

        train_x = data[train_indices]
        valid_x = data[valid_indices]

        train_y = a[train_indices]
        valid_y = a[valid_indices]

        train_x.shape , valid_x.shape, train_y.shape, valid_y.shape
```

```
Out[7]: ((10608, 64, 64, 1), (2652, 64, 64, 1), (10608, 3), (2652, 3))
```

FIG 5 : Splitting the data

Now, to start with the Neural Network used for Image Classification is Keras Sequential Model.

```
In [8]: from tensorflow import keras

        from tensorflow.keras.models import Sequential
        from tensorflow.keras.layers import Dense, Dropout, Flatten
        from tensorflow.keras.layers import Conv2D, MaxPooling2D

        #create a model for the image classifier

        num_output_classes = 3
        input_img_size = (64, 64, 1)  # 64x64 image with 1 color channel

        model = Sequential()
        model.add(Conv2D(32, kernel_size=(3, 3), activation="relu", input_shape=input_img_size))
        model.add(Conv2D(64, (3, 3), activation="relu"))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Dropout(0.25))
        model.add(Flatten())
        model.add(Dense(64, activation="relu"))
        model.add(Dropout(0.5))
        model.add(Dense(num_output_classes, activation="softmax"))
        model.compile(
            loss=keras.losses.categorical_crossentropy,
            optimizer=keras.optimizers.Adadelta(),
            metrics=["accuracy"],
        )
```

FIG 6 : Model with 8 layers

Here, the type of model used is sequential. A sequential model helps stack layers one on top of the other.

Layers *early* in the network architecture (i.e., closer to the actual input image) learn *fewer* convolutional filters while layers *deeper* in the network (i.e., closer to the output predictions) will learn *more* filters.

Conv2D layers in between will learn more filters than the early Conv2D layers but fewer filters than the layers closer to the output

I start with a total of 32 filters and an activation of ReLU - Rectified Linear Unit. It is an activation function used for transforming the summed weight input from the node into the activation of the node or output for that input. It is a piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero. It has become the default activation function for many types of neural networks because a model that uses it is easier to train and often achieves better performance.

The next layer consists of 64 filters and is followed by Max pooling to reduce the spatial dimensions of the output.

The next layer is a dropout layer. The Dropout layer randomly sets input units to 0 with a frequency of rate at each step during training time, which helps prevent overfitting. It is a technique where randomly selected neurons are ignored during training. They are “dropped-out” randomly. This means that their contribution to the activation of downstream neurons is temporarily removed on the forward pass and any weight updates are not applied to the neuron on the backward pass.

The flatten layer flattens the input and has no effect on batch size. Flattening a tensor means to remove all of the dimensions except for one. This is exactly what the Flatten layer do. A flatten operation on a tensor reshapes the tensor to have the shape that is equal to the number of elements contained in tensor non including the batch dimension.

Dense implements the operation:
output = activation(dot product (input, kernel) + bias)
where activation is the element-wise activation function passed as the activation argument, kernel is a weights matrix created by the layer, and bias is a bias vector created by the layer.
The next layer is a dense layer consisting of 64 filters.

The final Dense layer learns 3 filters with a softmax activation. Softmax is an activation function because it not only maps our output to a [0,1] range but also maps each output in such a way that the total sum is 1. The output of Softmax is therefore a probability distribution.

As our output spatial volume is *decreasing* our number of filters learned is *increasing*.

```
model.summary()
model.layers
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 62, 62, 32)        320
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 60, 60, 64)        18496
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 30, 30, 64)        0
_________________________________________________________________
dropout (Dropout)            (None, 30, 30, 64)        0
_________________________________________________________________
flatten (Flatten)            (None, 57600)             0
_________________________________________________________________
dense (Dense)                (None, 64)                3686464
_________________________________________________________________
dropout_1 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_1 (Dense)              (None, 3)                 195
=================================================================
Total params: 3,705,475
Trainable params: 3,705,475
Non-trainable params: 0
_________________________________________________________________
```

```
ıt[9]: [<tensorflow.python.keras.layers.convolutional.Conv2D at 0x7ff99d7fa220>,
 <tensorflow.python.keras.layers.convolutional.Conv2D at 0x7ff9827fc220>,
 <tensorflow.python.keras.layers.pooling.MaxPooling2D at 0x7ff98288d220>,
 <tensorflow.python.keras.layers.core.Dropout at 0x7ff9828c2610>,
 <tensorflow.python.keras.layers.core.Flatten at 0x7ff9827fce80>,
 <tensorflow.python.keras.layers.core.Dense at 0x7ff9828d8a00>,
 <tensorflow.python.keras.layers.core.Dropout at 0x7ff9828d8af0>,
 <tensorflow.python.keras.layers.core.Dense at 0x7ff9828edc40>]
```

FIG 7 : This tells the layers, their inputs and where they are located

Next the model is trained.

```
In [10]: #fitting the model by training it on the training dataset

batch_size = 128
epochs = 20

history  = model.fit(train_x, train_y,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(valid_x,valid_y))
```

```
Epoch 1/20
83/83 [==============================] - 91s 1s/step - loss: 15.7731 - accuracy: 0.5261 - v
al_loss: 4.7765 - val_accuracy: 0.7945
Epoch 2/20
83/83 [==============================] - 125s 2s/step - loss: 7.8522 - accuracy: 0.6791 - v
al_loss: 2.0397 - val_accuracy: 0.7986
Epoch 3/20
83/83 [==============================] - 193s 2s/step - loss: 4.8198 - accuracy: 0.6802 - v
al_loss: 0.7206 - val_accuracy: 0.8002
Epoch 4/20
```

FIG 8 : Now training the model by fitting it for 20 epochs (iterations)

When the epochs are completed, now i want to check how efficient my model really is. For that I would be plotting graphs for the model's performance on both the training and test sets for the following parameters - Loss, Accuracy, F-score, Precision and Recall.

Doing so gives the following graphs

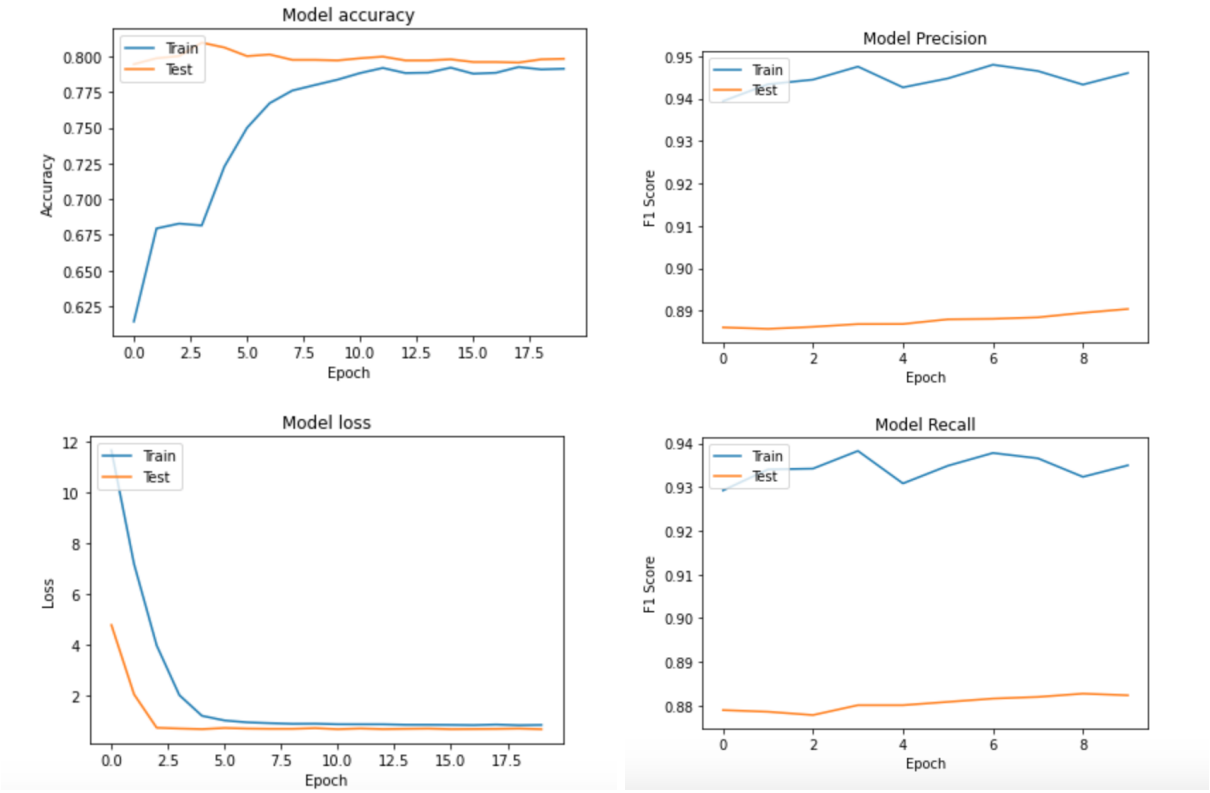


FIG 9 : Performance Parameters of the first model

As can be seen from the accuracy and loss graphs, the model is good.

Accuracy is one metric for evaluating classification models. It is the fraction of predictions our model got right. Accuracy = Number of correct predictions/ Total number of predictions.

The Loss Function is one of the important components of Neural Networks. Loss is a prediction error of the Neural Net. And the method to calculate the loss is called LossFunction. Loss is used to calculate the gradients. And gradients are used to update the weights of the Neural Net.

Precision fis the number of true positives (i.e. the number of items correctly labelled as belonging to the positive class) divided by the total number of elements labelled as belonging to the positive class (i.e. the sum of true positives and false positives). It is the degree of refinement with which an operation is performed.

When the recall is high, it means the model can classify all the positive samples correctly as Positive. Thus, the model can be trusted in its ability to detect positive samples.

The F1 score can be interpreted as a weighted average of the precision and recall, where an F1 score reaches its best value at 1 and worst at 0.

Next, the model now tells the values for the same in general:

```
In [60]: loss, accuracy, f1_score, precision, recall

Out[60]: (0.2730911374092102,
          0.8819758892059326,
          0.8812786936759949,
          0.8852399587631226,
          0.8774742484092712)
```

FIG 10 : Performance metric measure

Now that our model has been trained, it can predict values for a new dataset of images.

Let's get a classification report in order to see these metrics individually for each of the three classes.

Out[136]:

	precision	recall	f1-score	support
Healthy (Class 0)	0.926546	0.954959	0.940538	2087.000000
Pre-existing condition (Class 1)	0.722000	0.732252	0.727090	493.000000
Effusion/Mass (Class 2)	0.000000	0.000000	0.000000	72.000000
accuracy	0.887632	0.887632	0.887632	0.887632
macro avg	0.549515	0.562404	0.555876	2652.000000
weighted avg	0.863366	0.887632	0.875324	2652.000000

FIG 11 : Performance metrics of individual classes of the data

From the above table we can see that Class 0 has the highest number of samples and class 2 has the lowest (only 71). Moreover, all the metrics for Class 2 are extremely zero. This is due to highly imbalanced data being present.

Now that this model has been trained, it can now be used to label the new data.

The new labels come to be like -

In [30]:
```
labels = np.argmax(prediction, axis=-1)
print(labels)
```

```
[0 0 0 2 0 1 1 0 1 0 0 2 0 0 0 0 0 0 1 1 1 1 1 1 2 0 0 0 0 1 0 0 1 1 0 0 0
 0 0 1 0 1 0 0 0 1 0 2 1 1 0 0 0 0 0 1 0 1 1 1 1 1 0 0 1 0 0 0 2 1 0 1 0 1
 1 0 0 0 0 0 0 2 0 1 0 1 1 0 1 0 1 0 0 1 0 1 0 0 1 0 0 0 0 0 1 1 1 1 1 1 2
 1 1 1 0 1 1 0 0 0 1 1 0 0 1 1 1 1 0 1 1 1 0 1 1 1 1 1 0 0 1 0 0 0 1 0 0 1
 1 1 0 1 0 1 0 1 0 0 1 0 0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 1 0 1 2
 1 0 0 1 1 1 1 0 0 1 0 0 0 0 0 0 0 1 0 0 1 0 0 2 1 0 0 1 0 0 0 1 1 1 1 1 0
 1 0 0 1 1 0 1 0 0 0 0 1 1 0 0 1 1 0 1 1 1 0 1 0 0 1 1 1 0 0 0 0 1 0 1 0 1
 0 1 1 1 1 0 0 0 1 1 0 1 1 1 1 1 1 0 0 0 0 1 1 0 0 0 1 0 0 1 0 1 1 0 0 1 0
 0 1 1 1]
```

FIG 12 : The labels

Now, while this model gave me good scores for all metrics, I still wanted to measure other models before finalising this one.

Here's another model made with

```
#create a model for the image classifier

num_output_classes = 3
input_img_size = (64, 64, 1)  # 64x64 image with 1 color channel
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation="relu", input_shape=input_img_size))
#model.add(Conv2D(64, (3, 3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2, 2)))
#model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(64, activation="relu"))
model.add(Dropout(0.5))
model.add(Dense(num_output_classes, activation="softmax"))
model.compile(
    loss=keras.losses.categorical_crossentropy,
    optimizer=keras.optimizers.Adadelta(),
    metrics=["accuracy"],
)
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 62, 62, 32)        320
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 31, 31, 32)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 30752)             0
_________________________________________________________________
dense_2 (Dense)              (None, 64)                1968192
_________________________________________________________________
dropout_2 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_3 (Dense)              (None, 3)                 195
=================================================================
Total params: 1,968,707
Trainable params: 1,968,707
Non-trainable params: 0
_________________________________________________________________
```

```
[<tensorflow.python.keras.layers.convolutional.Conv2D at 0x7fb6e1608430>,
 <tensorflow.python.keras.layers.pooling.MaxPooling2D at 0x7fb6e16084c0>,
 <tensorflow.python.keras.layers.core.Flatten at 0x7fb6e15bcd00>,
 <tensorflow.python.keras.layers.core.Dense at 0x7fb6dd6b5460>,
 <tensorflow.python.keras.layers.core.Dropout at 0x7fb6e1424400>,
 <tensorflow.python.keras.layers.core.Dense at 0x7fb6e16d6af0>]
```

FIG 13 : A second model, with 6 layers

As can be seen, this is a model with 6 layers. Upon training this model and finding its metrics:

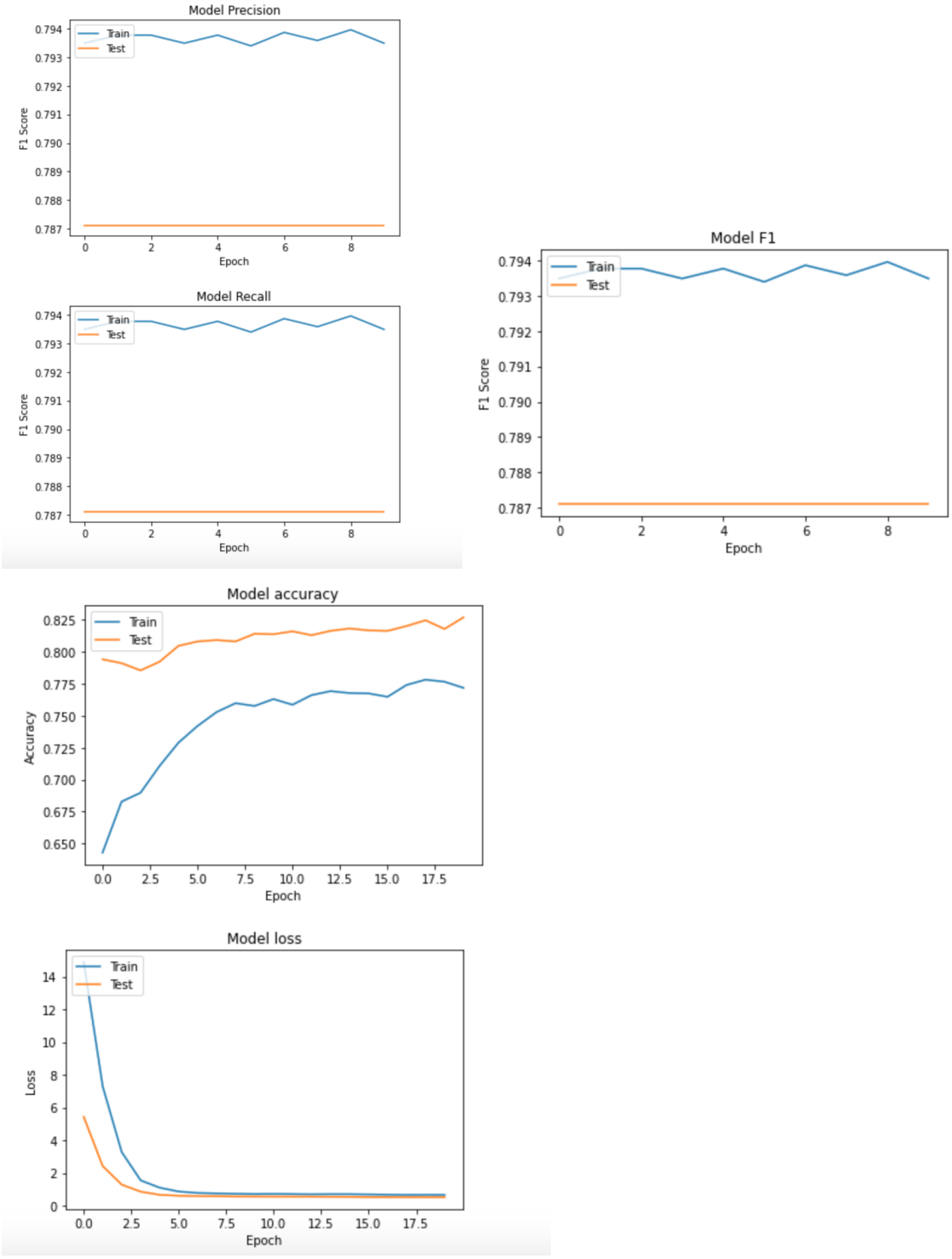


FIG 14 : Graphs of Model 2's parameter metrics

Metrics for this model are :

```
In [15]: loss, accuracy, f1_score, precision, recall
Out[15]: (0.3748573660850525,
          0.7869532704353333,
          0.7871127128601074,
          0.7871127724647522,
          0.7871127724647522)
```

FIG 15 : Model 2's parameters

We see that the loss for this model is greater than the first model.
Moreover, all remaining metrics are lower than the first model's.

So, till now, the first model is the better option.

Looking at the second model's classification report-

Out[13]:

	precision	recall	f1-score	support
Healthy (Class 0)	0.839190	0.972688	0.901021	2087.000000
Pre-existing condition (Class 1)	0.700000	0.326572	0.445367	493.000000
Effusion/Mass (Class 2)	0.000000	0.000000	0.000000	72.000000
accuracy	0.826169	0.826169	0.826169	0.826169
macro avg	0.513063	0.433087	0.448796	2652.000000
weighted avg	0.790531	0.826169	0.791854	2652.000000

FIG 16 : Individual class parameter metrics/classification report

We see that all metrics for class 2 are zero and they are comparatively lower than the first model's.

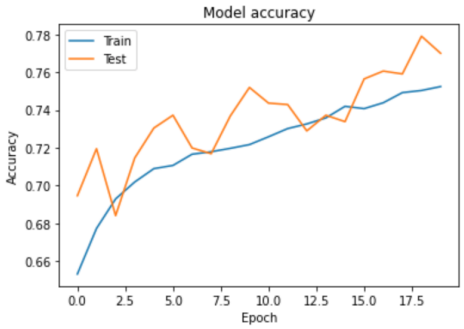
Similarly, another model I tested this against contains 4 layers only.

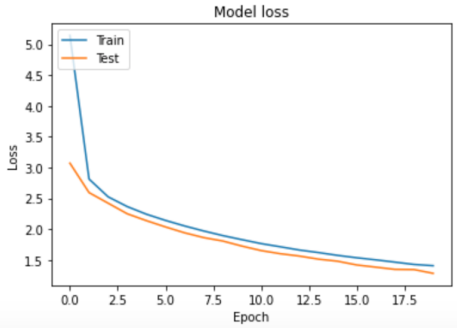
```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 62, 62, 32)        320
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 31, 31, 32)        0
_________________________________________________________________
flatten (Flatten)            (None, 30752)             0
_________________________________________________________________
dense_8 (Dense)              (None, 3)                 92259
=================================================================
Total params: 92,579
Trainable params: 92,579
Non-trainable params: 0
_________________________________________________________________
```

```
7]: [<tensorflow.python.keras.layers.convolutional.Conv2D at 0x7fc1a9c45610>,
     <tensorflow.python.keras.layers.pooling.MaxPooling2D at 0x7fc1a355ba00>,
     <tensorflow.python.keras.layers.core.Flatten at 0x7fc1a355b430>,
     <tensorflow.python.keras.layers.core.Dense at 0x7fc1a355cd90>]
```

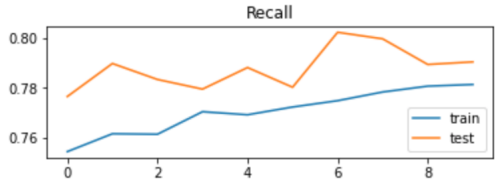
FIG 17 : Model 3 with 4 layers

Further, its graphs are

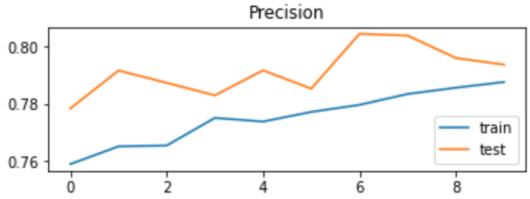




Out[34]: <matplotlib.legend.Legend at 0x7fc17623d730>



Out[35]: <matplotlib.legend.Legend at 0x7fc17618beb0>



Out[36]: <matplotlib.legend.Legend at 0x7fc17626a8e0>

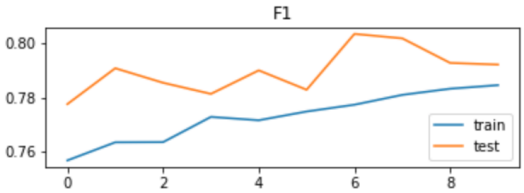


FIG 18 : Model 3 parameter metrics graphs

Metrics of the same -

```
In [11]: loss, accuracy, f1_score, precision, recall

Out[11]: (0.9922230243682861,
          0.7933635115623474,
          0.7920954823493958,
          0.793750524520874,
          0.7905013561248779)
```

FIG 19 : Model 3 parameter metrics

So the loss is the highest in this third model with other parameters comparatively weaker.

Classification report -

Out[12]:

	precision	recall	f1-score	support
Healthy (Class 0)	0.896670	0.863679	0.879865	2120.000000
Pre-existing condition (Class 1)	0.469775	0.590022	0.523077	461.000000
Effusion/Mass (Class 2)	0.032258	0.014085	0.019608	71.000000
accuracy	0.793363	0.793363	0.793363	0.793363
macro avg	0.466234	0.489262	0.474183	2652.000000
weighted avg	0.799320	0.793363	0.794813	2652.000000

FIG 20 - Classification report

We see now that Class 0 still has an alright score. Class 2 too has a few readings but Class 1 has suffered drastically.

Hence, post comparing the models as per their metrics - FScore, Precision, Recall, the first model gave the best readings for the validation data and hence a new test data has also been run on it to classify the images.

Conclusion:

So, 3 models have been compared on the basis of their performance metrics. We see that due to highly imbalanced data(class 2 having the least amount of samples), data augmentation is a path that can boost the model performance. But augmentation should be such that there is no loss of data but rather new data is added on the basis of class 2 samples already present.

I also showed which model is the best to use *the first one with 8 layers)

[1]https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/

[2]https://towardsdatascience.com/deep-learning-optimizers-436171c9e23f
[3]https://www.tensorflow.org/guide/keras/train_and_evaluate
[4]https://towardsdatascience.com/simple-guide-to-hyperparameter-tuning-in-neural-networks-3fe03dad8594
[5]https://www.alibabacloud.com/blog/part-4-image-classification-using-features-extracted-by-transfer-learning-in-keras_595292
[6]https://www.pyimagesearch.com/2018/12/31/keras-conv2d-and-convolutional-layers/

DATA AVAILABLE AT
https://www.kaggle.com/c/usc-dsci552-section-32415d-spring-2021-ps4/data

CODE AVAILABLE AT
https://github.com/usc-dsci552-32415D-spring2021/problem-set-04-AnanyaSharma25/tree/main