horizontal line

**Vanshika Gupta - 16CV245**

**Dibyadarshan Hota - 16CO154**

COVID-19 Radiography Database

**10th June 2020**

In case of queries, contact: [dibyadarshan.hota@gmail.com](mailto:dibyadarshan.hota@gmail.com), vanshika421@gmail.com

# OVERVIEW

As a part of the project, we have attempted to build a robust classification model for the COVID-19 radiography database. All of our attempts, results, and conclusions are documented hereby. This [**drive link**](https://drive.google.com/drive/folders/150RVRh4scJ0D2mbQliI8UK13rQGtDVH2) contains all the required codes, models, screenshots, requirements, heat maps, etc.

## Instructions for testing trained model

For testing the **separablenet.hdf5** (one of the top-performing models, code in file **covid.ipynb**)the following preprocessing needs to be done on X\_test.

1. Resize the images as follows: reshaped[i] = cv2.resize(c[i], (256, 256), interpolation = cv2.INTER\_CUBIC). Note: For best results interpolation must be cv2.INTER\_CUBIC only, this snippet is present in def reshape().
2. Normalize the input as follows: X\_train = (X\_train.astype('float32')) / 255.0. Note that normalization must be done **after** reshaping.
3. The Y\_test must be converted back from one-hot representation to integer class labels, refer to def display\_metrics(model).

## Instructions for running covid.ipynb

Here are few instructions needed to run the file locally:

1. All the requirements at the bottom of this doc must be imported/installed.
2. In the def load\_drive() part, set the base folder path.
3. In the “Load Data” part, set the path to the .npy file.
4. In def train(model, save\_name) the save\_path must be set appropriately. The model.fit callback saves the model weights which has the highest validation accuracy. **For evaluating the model, load\_model with the highest val\_accuracy saved by the callback instead of using the final model, this is a must for optimal results**. **This is necessary since we have used the holdout method instead of k fold validation.**

# METHODOLOGY

1. To take care of the imbalanced dataset we have oversampled the rare class using [imgaug](https://imgaug.readthedocs.io/en/latest/).
2. For making the model robust we have used [ImageDataGenerator](https://keras.io/api/preprocessing/image/) for online augmentation.
3. We have experimented and used different techniques like dropout, batch normalization, regularization, early stopping, and class weighing for getting the best performance.
4. We have also attempted transfer learning and fine-tuning in some of the models.

# ATTEMPTS

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Sl. No** | **Data Used**  (All details) | **Model Function** | **Model** | **Training Accuracy**  (at highest val\_acc) | **Validation Accuracy** | **Epochs** |
| 1 | No augmentation + 70-30 split | def simpleCNN() | Simple Deep CNN | 0.9272 | 0.9393 | 10 |
| 2 | Oversampling (+1x) + VGG on 4 layers + 0.19 test + Adam amsgrad + lr = 0.0002. No class weight | def separable\_model(image\_shape=(256, 256,3)) | Separable with a partial transfer learning | 0.967 | 0.9637 | 20 |
| 3 | Class weights (weight 6 for rare) and no oversampling + ^ above | def separable\_model(image\_shape=(256, 256,3)) | Separable with a partial transfer learning | 0.9435 | 0.9705 | 20 |
| 4 | Oversampling (+1x) + class weight | def alexnetv2\_model(image\_shape=(256, 256, 3)) | AlexNet architecture with separable convolution | 0.954 | 0.96825 | 20 |
| 5 | Oversampling (+1x) + class weight (1.2 for rare) + Adam with amsgrad and lr = 0.0004 | def separablenet\_model(image\_shape=(256, 256, 3)) | Custom model with separable convolution and transfer learning | 0.97 | 0.9818 | 50 |
| 6 | Oversampling (+2x) + augmentation(double) + 0.18 test + Adam | def denseNet(image\_shape=(256, 256, 3)) | DenseNet | 0.9837 | 0.9378 | 20 |
| 7 | def SqueezeNet(include\_top=True, weights=None, input\_tensor=None, input\_shape=(256,256,3),  pooling=None, classes=3) | SqueezeNet | 0.9014 | 0.91399 |
| 8 | def alexnet\_model(img\_shape=(256, 256, 3), n\_classes=3, l2\_reg=0., weights=None): | AlexNet | 0.9725 | 0.96890 |
| 9 | Oversampling (+2x) + augmentation(double) + 0.18 test + Adam + lr = 0.0005 | def resNet50(image\_shape = (256,256,3)): | ResNet | 0.9559 | 0.9617 | 20 |
| 10 | Oversampling (+2x) + augmentation(double) + 0.18 test + Adam + lr = 0.0005 | def SqueezeNet(include\_top=True, weights=None, input\_tensor=None, input\_shape=(256,256,3),  pooling=None, classes=3) | SqueezeNet | 0.9508 | 0.95455 | 20 |
| 11 | Oversampling (+2x) + augmentation(double) + 0.18 test + Adam + lr = 0.0005 | def SqueezeNet(include\_top=True, weights=None, input\_tensor=None, input\_shape=(256,256,3),  pooling=None, classes=3) | SqueezeNet | 0.9470 | 0.9545 | 30 |

These are some of the top models we documented. For each model, we tried a number of different things. We varied a lot of hyperparameters to get these results. We varied dropout rates, learning rates, class weights, etc.

# NOVELTY

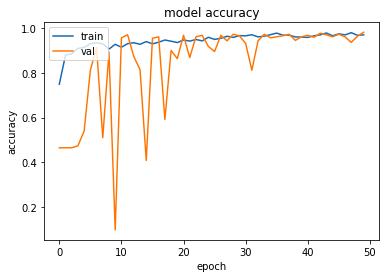
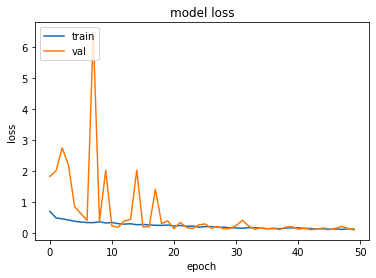
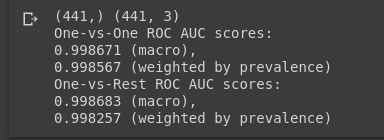
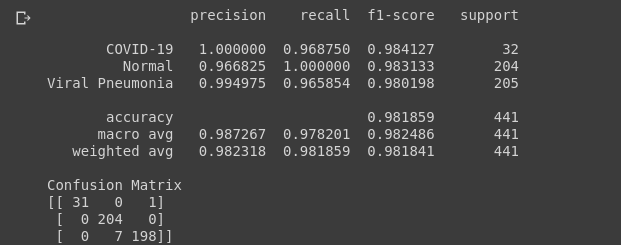
In this section, we focus on our custom model, it is named as def separable\_model() in the code. Few things to note are:

* We initialized the first few layers from the pre-trained ImageNet VGG16 model. This is because the first few layers capture very general details common to a lot of datasets like patches, edges, etc. Instead of randomly initialized weights for these layers, it is certainly better to have them defined using ImageNet.
* Xception model ([paper](https://arxiv.org/abs/1610.02357)) makes use of separable convolutions, inspired by that, we have done the same. Separable convolutions help in reducing the number of parameters significantly and help in constructing relatively larger models. The architecture is as follows:

1. # Layer 1
2. separablenet.add(Conv2D(64, (3, 3), input\_shape=image\_shape, padding='same'))
3. separablenet.add(Activation('relu'))
4. separablenet.add(BatchNormalization())
5. separablenet.add(MaxPooling2D(pool\_size=(2, 2)))
6. # Layer 2
7. separablenet.add(SeparableConv2D(128, (3, 3), padding='same'))
8. separablenet.add(Activation('relu'))
9. separablenet.add(BatchNormalization())
10. separablenet.add(MaxPooling2D(pool\_size=(2, 2)))
11. # Layer 3
12. separablenet.add(ZeroPadding2D((1, 1)))
13. separablenet.add(SeparableConv2D(256, (3, 3), padding='same'))
14. separablenet.add(Activation('relu'))
15. separablenet.add(BatchNormalization())
16. separablenet.add(MaxPooling2D(pool\_size=(2, 2)))
17. # Layer 4
18. separablenet.add(ZeroPadding2D((1, 1)))
19. separablenet.add(SeparableConv2D(512, (3, 3), padding='same'))
20. separablenet.add(Activation('relu'))
21. separablenet.add(BatchNormalization())
22. # Layer 5
23. separablenet.add(ZeroPadding2D((1, 1)))
24. separablenet.add(SeparableConv2D(512, (3, 3), padding='same'))
25. separablenet.add(Activation('relu'))
26. separablenet.add(BatchNormalization())
27. separablenet.add(MaxPooling2D(pool\_size=(2, 2)))
28. # Layer 6
29. separablenet.add(Flatten())
30. separablenet.add(Dense(1024))
31. separablenet.add(Activation('relu'))
32. separablenet.add(BatchNormalization())
33. separablenet.add(Dropout(0.8))
34. # Layer 7
35. separablenet.add(Dense(512))
36. separablenet.add(Activation('relu'))
37. separablenet.add(BatchNormalization())
38. separablenet.add(Dropout(0.8))
39. # Layer 8
40. separablenet.add(Dense(3))
41. separablenet.add(BatchNormalization())
42. separablenet.add(Activation('softmax'))

* Note that we have used BN after ReLu unlike the original paper (see observation).
* We have used both oversampling and class weights. The reason for this is, to handle a highly imbalanced dataset, oversampling is a common technique but oversampling a lot leads to overfitting (since augmentations are a bit limited). To make up for the rest, we have made use of class weights.

Results:

1. 
2. 
3. 
4. 

# OBSERVATION

1. The ordering of batch normalization and ReLu according to the original paper was as follows, -> CONV/FC -> BatchNorm -> ReLu (or other activation) -> Dropout -> CONV/FC ->. But according to [sources](https://github.com/ducha-aiki/caffenet-benchmark/blob/master/batchnorm.md), BN must be after activation. We tried both the variants and realized placing BN after activation gives slight improvement.
2. Adam outperforms RMSprop. AMSgrad as mentioned in [this paper](https://openreview.net/pdf?id=ryQu7f-RZ) has proof of convergence, hence we used that, the results obtained were quite satisfactory.
3. Relatively simpler or shallow models gave a better performance as compared to very deep models. The reason for this must be the vanishing gradient problem.
4. Higher oversampling led to overfitting and high class weight distorted the training loss. For optimal results, we ended up using both, the weights and count of oversampled images were another hyperparameter.
5. Online augmentation was useful due to the large image size/dataset as it otherwise resulted in the overflow of RAM.

# REQUIRED VERSIONS

The ‘**requirements.txt’** files have been attached in the root folder.

‘**Requirements\_all.pdf’** lists all the dependencies.

‘**Requirements\_main.pdf’** lists the primary dependencies.

**Note:** For running the ‘GradCam’, **scipy==1.1.0** is used instead of **1.4.1**