BU CS440 CLASS CHALLENGE REPORT

Name: Tegveer Ghura

TASK 1

Although comparing two different models was not required for this task, I wanted to explore more about performance of optimizers, given the same pre-trained model (VGG16 with ImageNet weights), in a binary classification task. Therefore, I used two different optimizers, SGD+momentum+nesterov and Adagrad, to obtain two different results for the same pre-trained model. SGD or Stochastic Gradient Descent is one of the most basic gradient descent techniques available on Keras and is **not adaptive,** which means it uses a global and equal learning rate for all parameters. Therefore, all of our parameters are being updated with a constant factor. However, the Adagrad optimizer is adaptive, which adaptively tunes the learning rate throughout the training phases and knows which direction to accelerate and decelerate in. Moreover, few years back a UC Berkeley paper was published and, in this paper, the authors compared adaptive optimizers (Adam, RMSprop and AdaGrad) with SGD, observing that SGD had better generalization than adaptive optimizers. Therefore, these papers and blogs/journals prompted me to conduct my own little experiment to compare optimizers, given the same pre-trained model. In the following pages I distinguish in-depth between the performances of both models and conclude by analyzing their respective test accuracy and test loss.

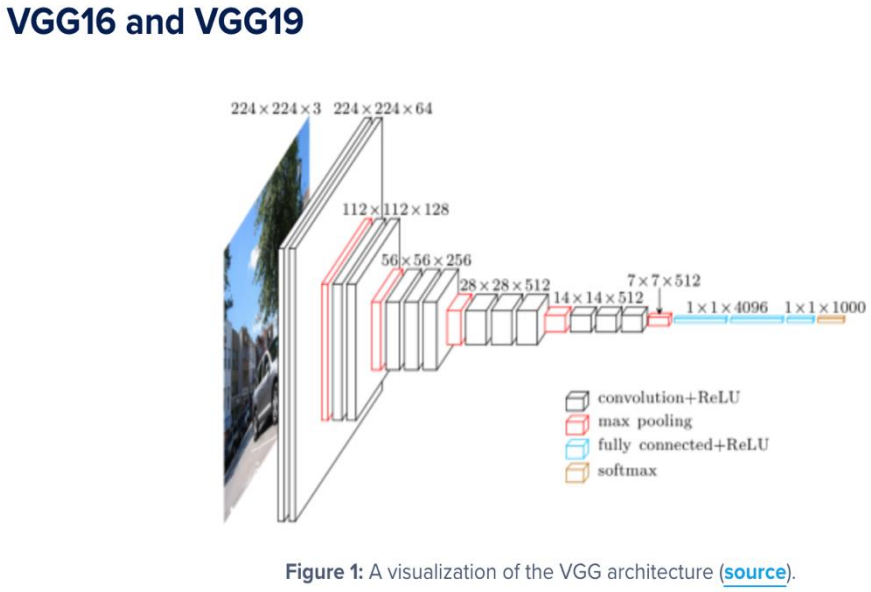
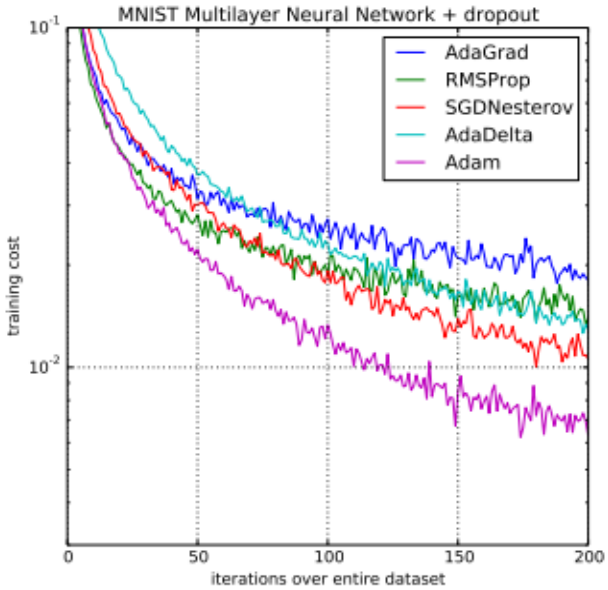


Figure 1: A visualization of the VGG architecture (source).



Picture on the Left: https://www.pyimagesearch.com/2017/03/20/imagenet-vggnet-resnet-inception-xception-keras/

Picture on the Right: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

BU CS440 CLASS CHALLENGE REPORT

Name: Tegveer Ghura

Detailed Description of the Model with SGD+nesterov+momentum Optimizer

Description of architecture used:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Functional)           (None, 7, 7, 512)         14714688
_________________________________________________________________
flatten (Flatten)            (None, 25088)             0
_________________________________________________________________
dropout (Dropout)            (None, 25088)             0
_________________________________________________________________
dense_feature (Dense)        (None, 256)               6422784
_________________________________________________________________
dropout_1 (Dropout)          (None, 256)               0
_________________________________________________________________
dense (Dense)                (None, 1)                 257
=================================================================
Total params: 21,137,729
Trainable params: 21,137,729
Non-trainable params: 0
_________________________________________________________________
```

The pre-trained network used is VGG16 with ImageNet weights for transfer learning. Specifically, for our VGG16 pre-trained network, we do not include the 3 fully-connected layers at the top of the network ("include_top=False"), but fit the VGG16 network to a series of densely-connected layers that we include ourselves. The VGG16 network also takes in the default input shape of (224,224,3), which corresponds to width and height of 224 and 3 input channels (channels_last data format). The first dense layer is a hidden layer with 256 units of output space and activation function ReLU (f(x)=max(0,x)). The second dense layer is our output layer with 1 output unit and a sigmoid activation function due to binary classification. Before we specify our first dense layer, we initialize a flatten layer, which prepares a vector from image data input for the fully connected layers. We also include two Dropout layers with a rate of 0.25 before and after the first dense layer to reduce overfitting by model averaging.

BU CS440 CLASS CHALLENGE REPORT

Name: Tegveer Ghura

Optimizer, loss function, parameters, and any regularization used:

The optimizer being used for comparison in this model is Stochastic gradient descent (SGD) that performs a parameter update for *each* training example $x^{(i)}$ and label $y^{(i)}$:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)}).$$

Instead of using a standard (vanilla) SGD optimizer, I have used an SGD+momentum+nesterov optimizer and I shall describe what the terms "momentum" and "nesterov" mean and how they contribute to the performance of the SGD optimizer. Because we are using mini-batches for training, the SGD optimizer can also be referred to as Mini-batch gradient descent, which does not guarantee convergence and has drawbacks that can slow down training time painfully. In order to tackle these drawbacks, such as adjustment of learning rates and getting trapped in several suboptimal local minima, we introduce momentum.

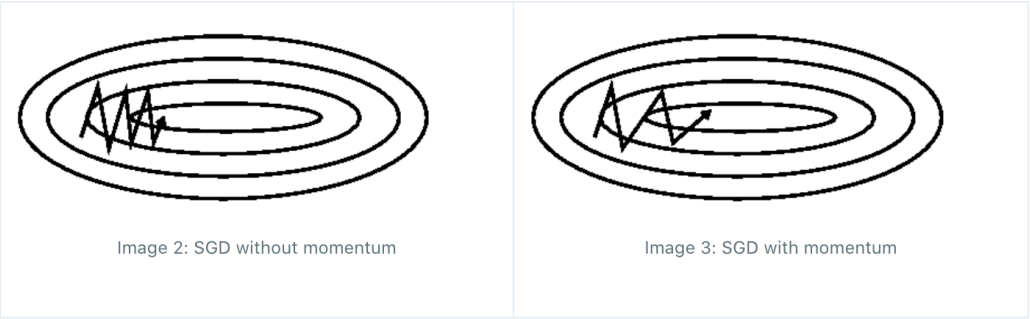


Image 2: SGD without momentum

Image 3: SGD with momentum

Momentum [5] is a method that helps accelerate SGD in the relevant direction and dampens oscillations as can be seen in Image 3. It does this by adding a fraction γ of the update vector of the past time step to the current update vector:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

For my model, I have set the momentum term, γ, to 0.9 (a common value). The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation.

BU CS440 CLASS CHALLENGE REPORT

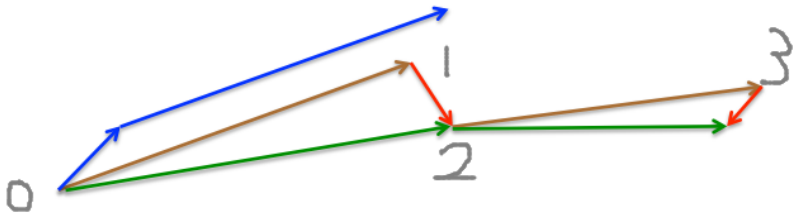
Name: Tegveer Ghura

Now, I shall explain the term "nesterov", which stands short for **Nesterov Accelerated Gradient** (NAG). NAG is a way to give our momentum term a kind of far-sightedness to get a rough idea of where our parameters are going to be in the future and, hence, know when to slow down before the hill slopes up again during gradient descent. We can now effectively look ahead by calculating the gradient not with respect to our current parameters θ but with respect to the approximate future position of our parameters.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

A picture of the Nesterov method

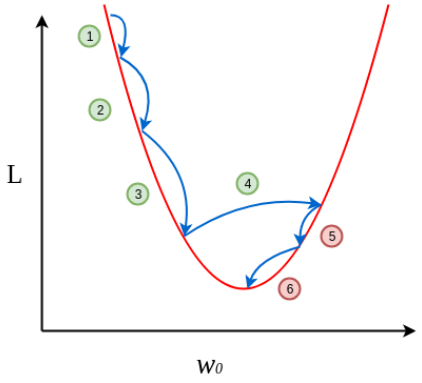
- First make a big jump in the direction of the previous accumulated gradient.
- Then measure the gradient where you end up and make a correction.



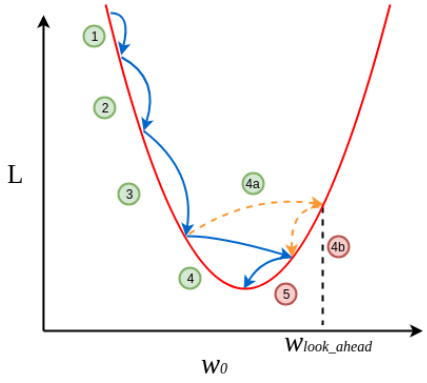
brown vector = jump, red vector = correction, green vector = accumulated gradient

blue vectors = standard momentum

Further explanation highlighting the difference between momentum and NAG methods:



(a) Momentum-Based Gradient Descent



(b) Nesterov Accelerated Gradient Descent

$\bigcirc \Longrightarrow \frac{\partial L}{\partial w_0} = \frac{Negative(-)}{Positive(+)}$ $\bigcirc \Longrightarrow \frac{\partial L}{\partial w_0} = \frac{Negative(-)}{Negative(-)}$

BU CS440 CLASS CHALLENGE REPORT

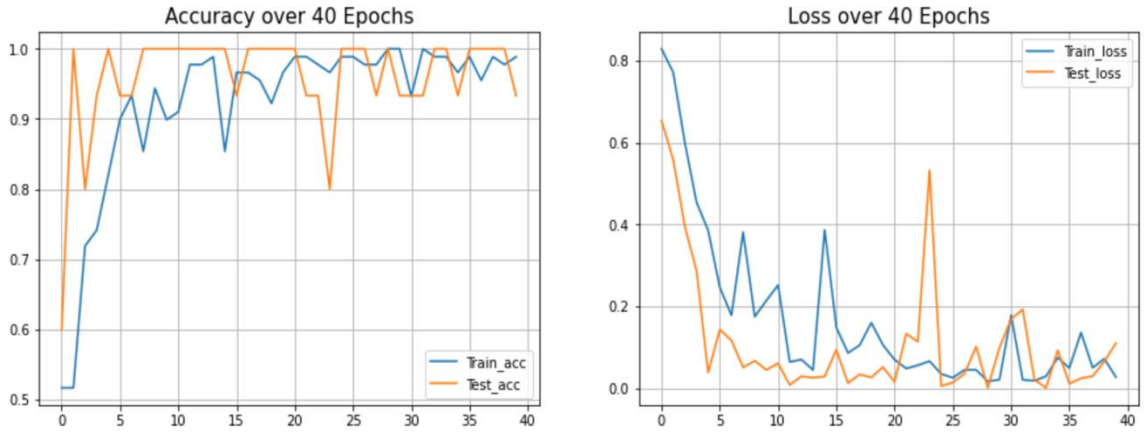
Name: Tegveer Ghura

The loss function used is binary cross entropy because our task is that of binary classification (two target classes: Covid-19 Pneumonia vs Normal). This is also a reason why we used the sigmoid activation function, instead of soft-max, in our second dense layer with 2 output units. Therefore:

$$\text{Cost}\left(h_\theta(x^{(i)}), y^{(i)}\right) = \begin{cases} -\log\left(h_\theta(x^{(i)})\right) & if\ y = 1 \\ -log\left(1 - h_\theta(x^{(i)})\right) & if\ y = 0 \end{cases} = -y^{(i)} \log h_\theta(x^{(i)}) - \left(1 - y^{(i)}\right) \log\left(1 - h_\theta(x^{(i)})\right)$$

Parameters used to fit the model include a batch size of 15, 40 training epochs with 6 steps per training epoch or 6 batches per training epoch, and 1 validation step or 1 batch per validation epoch. The learning rate for the optimizer is set at a low value of 0.0005 in order for the model to have highest chance of convergence at the expense of time (although our model converged fairly quickly due to sparse data). The regularization methods used are Dropout, which has previously been described, and Data Augmentation by using the ImageDataGenerator() class of Tensorflow. Data Augmentation here mainly serves as a preprocessing tool for our training and validation data. The arguments used in ImageDataGenerator() and their descriptions/purposes are as follows:
Rescaling images (rescale); standardizing pixel values across the entire dataset or also known as feature standardization (featurwise_center and featurewise_std_normalization); shifting width and height range of images (width_shift_range and height_shift_range); changing shear angle counter-clockwise and zooming in (shear_range and zoom_range); applying ZCA Whitening that reduces the redundancy in the matrix of pixel images, helping better highlight the structures and features in the image to the learning algorithm (zca_whitening); making the model more robust to variance in color (channel_shift_range); and randomly flipping the images horizontally as well as vertically (horizontal_flip and vertical_flip).

Plot and comment on the accuracy and the loss:

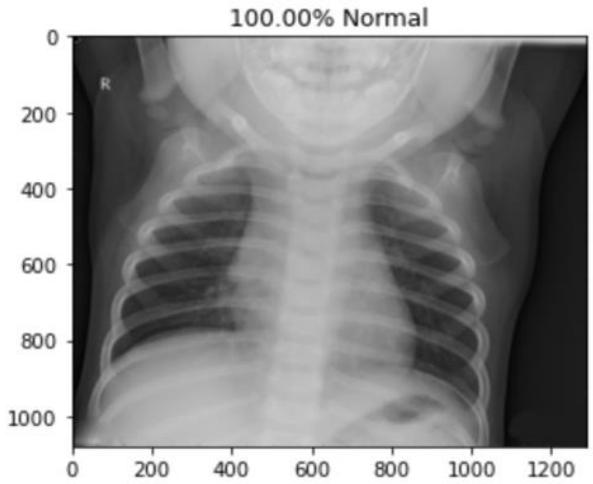


```
18/18 [==============================] - 5s 278ms/step - loss: 0.0097 - accuracy: 1.0000
Test loss: 0.009725017473101616
Test accuracy: 1.0
```

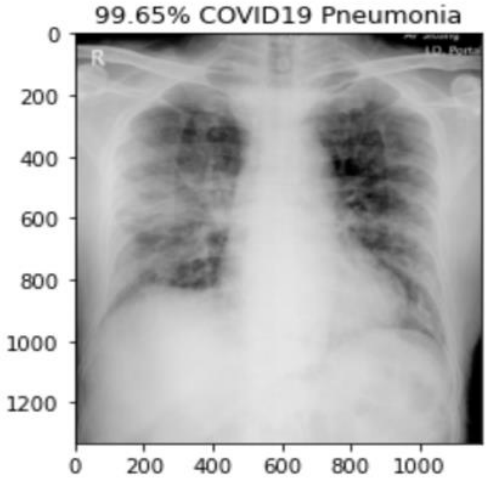
From the accuracy plot, we can see that our test accuracy quickly converges to almost 100% accuracy in and about 5-10 epochs and thereafter oscillates mostly between 0.933 (93.33%) and 1.0 (100%) accuracy. The same can be said for the test loss that converges to almost 0 in and about 5-10 epochs and thereafter oscillates mostly between 0 and 0.2 loss. Therefore, our model performs well on unseen images and generalizes well on those images too. More specifically, we get perfect training and test accuracy for the overall model as seen in the image at the top of this page. However, we must not forget that our dataset used was still very small for deep learning tasks and, on top of that, our task was that of to differentiate between only two sets of classes. As a result of this simplistic experiment, our model achieved such precise and accurate numbers. In order to get a better sense of our model performance, we could use a larger dataset or add in more sets of classes to differentiate between (multiclass classification task instead of binary classification), which we shall analyze in Task 2!

Moreover, we also plotted our test images with their respective predictions our model outputted. We have a total of 18 test images, with each set, Covid19 Pneumonia and Normal, consisting of 9 images. For all the 18 images, we obtained perfect results as all sets of images were classified correctly by our model with over 95% accuracy for all images! Few examples are posted below:

normal/NORMAL2-IM-1385-0001.jpeg



covid/ryct.2020200034.fig5-day4.jpeg

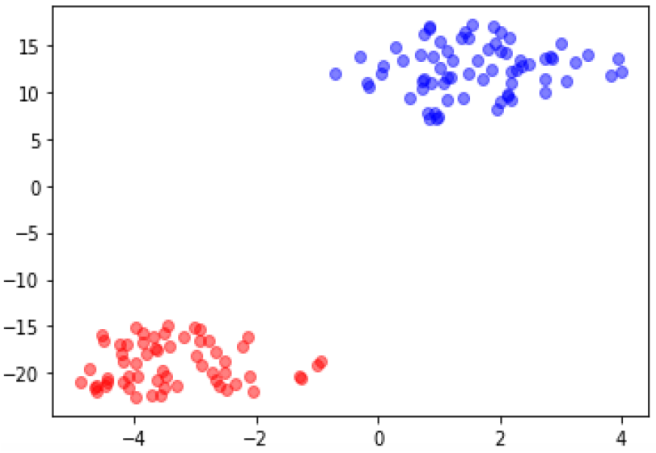


BU CS440 CLASS CHALLENGE REPORT

Name: Tegveer Ghura

Plot and comment on the t-SNE visualization:

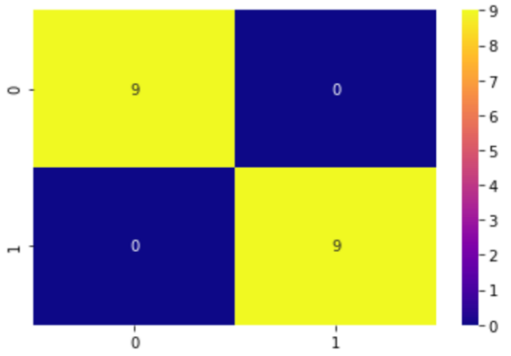
The TSNE plot for our model with SGD+momentum+nesterov optimizer displays separate, compact clusters of both classes, which means that the features extracted were good. A very clear distinction between the red (indicating Covid-19) and blue clusters (indicating Normal) is easily noticed, reinforcing the fact that our model generalizes very well on unseen data.



Plot and comment on the Confusion Matrix and Classification Report (extra):

We had previously seen that when we plotted our test images, we obtained perfect results. As a result, it was expected that the confusion matrix give this kind of output with 9 True Positives (correctly predicted Covid19 Pneumonia sets of images) and 9 True Negatives (correctly predicted Normal sets of images). Therefore, we see 0 False Positives and False Negatives and an F-1 measure of 1, a reinforcement of our earlier claim that our model performed perfectly on the given test images.

	precision	recall	f1-score	support
covid	1.00	1.00	1.00	9
normal	1.00	1.00	1.00	9
accuracy			1.00	18
macro avg	1.00	1.00	1.00	18
weighted avg	1.00	1.00	1.00	18



BU CS440 CLASS CHALLENGE REPORT

Name: Tegveer Ghura

Detailed Description of the Model with Adagrad Optimizer

Description of architecture used:

Just like the model with SGD+momentum+nesterov optimizer, I used the VGG16 pre-trained network with ImageNet weights for transfer learning and did not include the 3 fully-connected layers at the top of the network ("include_top=False"), but fit the VGG16 network to a series of densely-connected layers that we include ourselves. The VGG16 network also takes in the default input shape of (224,224,3), which corresponds to width and height of 224 and 3 input channels (channels_last data format). The first dense layer is a hidden layer with 256 units of output space and activation function ReLU (f(x)=max(0,x)). The second dense layer is our output layer with 1 output unit and a sigmoid activation function due to binary classification. Before we specify our first dense layer, we initialize a flatten layer, which prepares a vector from image data input for the fully connected layers. For this model with Adagrad Optimizer, I did not use any Dropout layer(s) because I wanted to obtain maximum training accuracy so that the model could try to give a similar, robust output like that of the model with SGD+momentum+nesterov optimizer.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Functional)           (None, 7, 7, 512)         14714688
_________________________________________________________________
flatten (Flatten)            (None, 25088)             0
_________________________________________________________________
dense_feature (Dense)        (None, 256)               6422784
_________________________________________________________________
dense (Dense)                (None, 1)                 257
=================================================================
Total params: 21,137,729
Trainable params: 21,137,729
Non-trainable params: 0
_________________________________________________________________
```

BU CS440 CLASS CHALLENGE REPORT

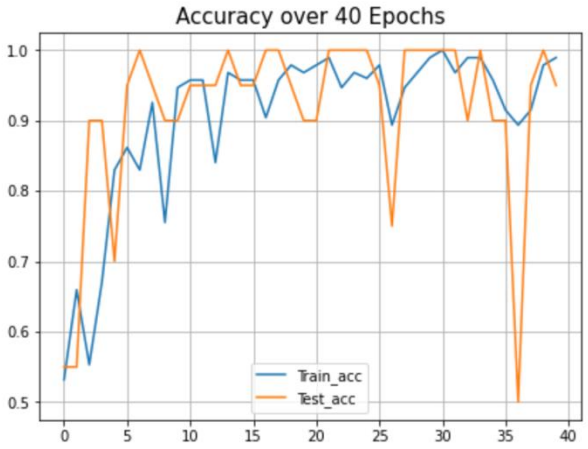
Name: Tegveer Ghura

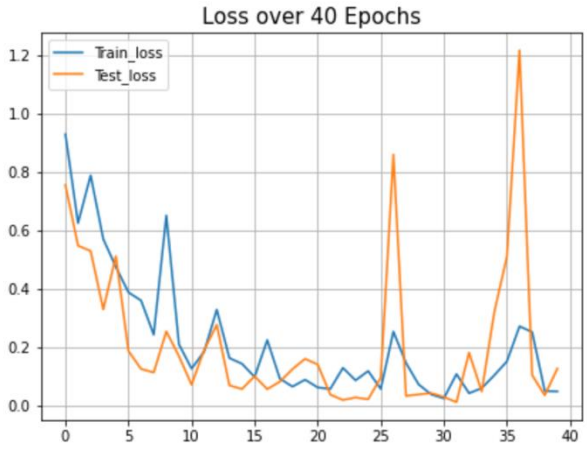
Optimizer, loss function, parameters, and any regularization used:

While introducing my outline for Task 1, I explained briefly that the Adagrad optimizer is adaptive, which means that our model can now adapt our updates to each individual parameter to perform larger or smaller updates depending on their importance (different learning rate for every parameter θ_i at every time step t). Specifically, Adagrad adapts the learning rate to the parameters, performing smaller updates (i.e. low learning rates) for parameters associated with frequently occurring features, and larger updates (i.e. high learning rates) for parameters associated with infrequent features. For this reason, it is well-suited for dealing with our experiment consisting of sparse data.

Note that for this model the optimizer, its learning rate, the exclusion of Dropout layers, and batch size are different from the previous model. The batch size used for this model is 10 but for the previous model it was 15. Again, the main reason to set this hyperparameter to 10 was to help the model converge quicker to match that level of convergence as the previous model. Otherwise number of epochs and data augmentation, as a regularization method, are the same as that of the previous model. Lastly, the Adagrad model took 4091.3 seconds to run through all epochs and finish model training, whereas the previous model took 3379.2 seconds. Although comparing overall training time of both models is not indicative of model performance and does not help us infer much, it can just be kept at the back of mind for future purposes.

Plot and comment on the accuracy and the loss



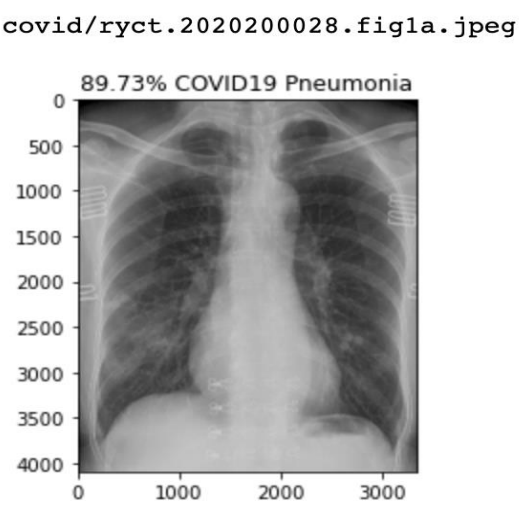


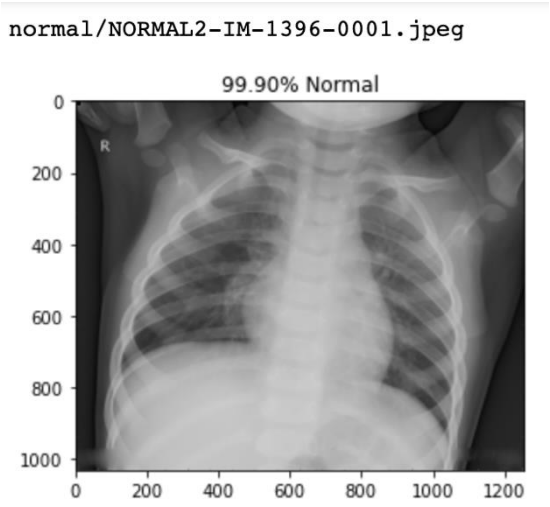

```
18/18 [==============================] - 5s 282ms/step - loss: 0.0298 - accuracy: 1.0000
Test loss: 0.029754225164651 87
Test accuracy: 1.0
```

BU CS440 CLASS CHALLENGE REPORT

Name: Tegveer Ghura

Compared to our previous model, it is noticed from the test accuracy plot that the model accuracy and loss vary more after the 25th epoch. Our previous model had a more robust performance after 25 epochs, as its test accuracy oscillated mainly between 0.933 and 1.0. Convergence for this model takes a few more epochs than the previous model. However, what matters is that this model performs well on unseen images and generalizes well on those images too, as we get perfect training and test accuracy for the overall model. Therefore, both models gave us perfect accuracy; however, the test and training loss for the latter model was a bit higher than that of the previous model, but such small differences can be negligible for facile tasks. Moreover, we also plotted our test images with their respective predictions our model outputted. We have a total of 18 test images, with each set, Covid19 Pneumonia and Normal, consisting of 9 images. For this model, however, accuracies for images were not as high as that of the previous model and the variation in accuracy and higher loss could explain that. What is important is that all images were classified correctly by our model, with most images belonging to the "Covid19 Pneumonia" class being classified with < 95% accuracy. Few examples with lowest accuracies are posted below:

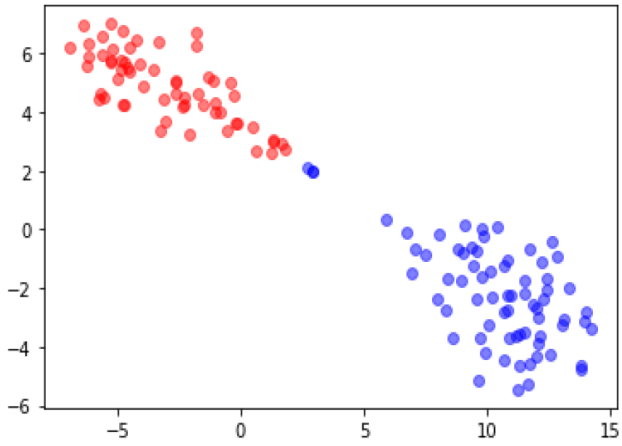




BU CS440 CLASS CHALLENGE REPORT

Name: Tegveer Ghura

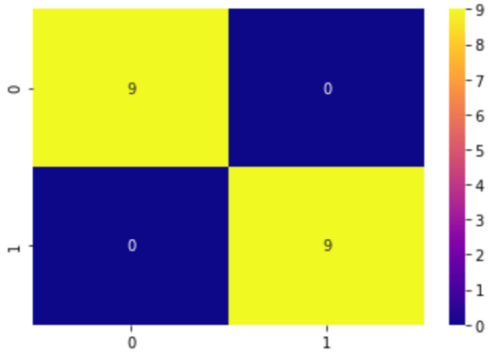
Plot and comment on the t-SNE visualization:



The TSNE plot for the Adagrad model does not do as well compared to the previous model. This claim can be backed by the fact that a few points of the Normal (blue) label are closer to the Covid-19 (red) cluster. Therefore, our previous model has a better performance in terms of features extracted.

Plot and comment on the Confusion Matrix and Classification Report (extra):

	precision	recall	f1-score	support
covid	1.00	1.00	1.00	9
normal	1.00	1.00	1.00	9
accuracy			1.00	18
macro avg	1.00	1.00	1.00	18
weighted avg	1.00	1.00	1.00	18



We had previously seen that when we plotted our test images, we obtained perfect results. As a result, it was expected that the confusion matrix give this kind of output with 9 True Positives (correctly predicted Covid19 Pneumonia sets of images) and 9 True Negatives (correctly predicted Normal sets of images). Therefore, we see 0 False Positives and False Negatives and an F-1 measure of 1, a reinforcement of our earlier claim that our model performed perfectly on the given test images.

BU CS440 CLASS CHALLENGE REPORT

Name: Tegveer Ghura

TASK 2

The two architectures being compared in this task are Xception and ResNetV2. My goal was to implement those architectures that were much smaller in size compared to VGG16/VGG19, which are painfully slow to train, but also obtain good, robust model performance overall. The Keras API references website describes a lot of features, such as size, top-k performance, and depth, of almost all available transfer learning architectures. The website helped me choose my two architectures for this task. It was surprising to notice that VGG16 and VGG19, being the largest architectures in size, had lower top-1 and top-5 accuracies on average compared to other architectures, like Xception and ResNetV2. Moreover, after having used VGG16 in Task 1 and attaining perfect accuracy using both optimizers, I felt a change was required to play around with other architectures. A few of the models and their performances from the Keras website are as follows:

Model	Size	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth
Xception	88 MB	0.790	0.945	22,910,480	126
VGG16	528 MB	0.713	0.901	138,357,544	23
VGG19	549 MB	0.713	0.900	143,667,240	26
ResNet50	98 MB	0.749	0.921	25,636,712	-
ResNet101	171 MB	0.764	0.928	44,707,176	-
ResNet152	232 MB	0.766	0.931	60,419,944	-
ResNet50V2	98 MB	0.760	0.930	25,613,800	-
ResNet101V2	171 MB	0.772	0.938	44,675,560	-
ResNet152V2	232 MB	0.780	0.942	60,380,648	-
InceptionV3	92 MB	0.779	0.937	23,851,784	159
InceptionResNetV2	215 MB	0.803	0.953	55,873,736	572
MobileNet	16 MB	0.704	0.895	4,253,864	88
MobileNetV2	14 MB	0.713	0.901	3,538,984	88
DenseNet121	33 MB	0.750	0.923	8,062,504	121
DenseNet169	57 MB	0.762	0.932	14,307,880	169
DenseNet201	80 MB	0.773	0.936	20,242,984	201

And given below are the visualized architectures of Xception and Resnet50 (original) vs ResNet50V2 (proposed). basic architecture of the post-activation (original version 1) and the pre-activation (version 2) of versions of ResNet.

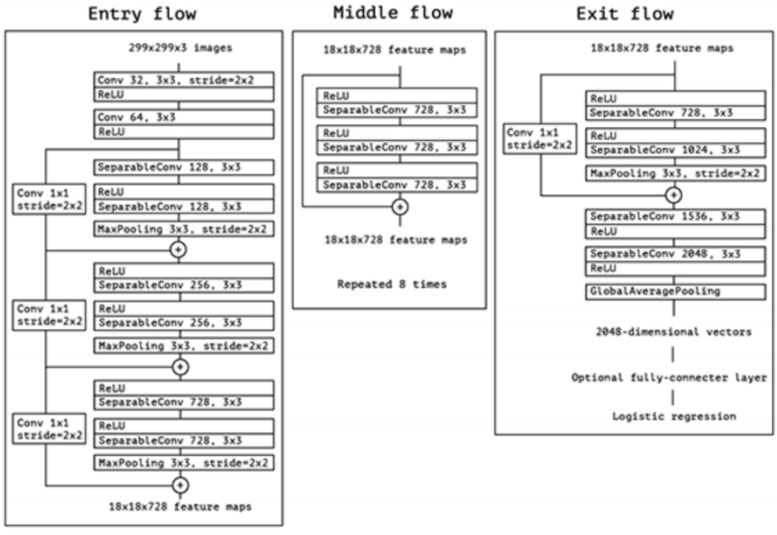


Figure 6: The Xception architecture.

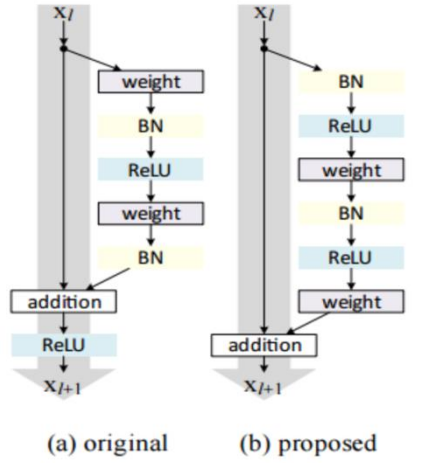


Figure 4: *(Left)* The original residual module. *(Right)* The updated residual module using pre-activation.

BU CS440 CLASS CHALLENGE REPORT

Name: Tegveer Ghura

Detailed Description of Xception Architecture:

Xception was proposed by none other than François Chollet himself, the creator and chief maintainer of the Keras library. Xception is an extension of the Inception architecture which replaces the standard Inception modules with depthwise separable convolutions. Xception sports the smallest weight serialization at only 91MB. As written in the original research paper: Xception is a novel deep convolutional neural network architecture inspired by Inception, where Inception modules have been replaced with depthwise separable convolutions. This architecture slightly outperforms Inception V3 on the ImageNet dataset (which Inception V3 was designed for), and significantly outperforms Inception V3 on a larger image classification dataset comprising 350 million images and 17,000 classes. Since the Xception architecture has the same number of parameters as Inception V3, the performance gains are not due to increased capacity but rather to a more efficient use of model parameters. Therefore, the name Xception stands for "Extreme Inception". The Xception architecture has 36 convolutional layers forming the feature extraction base of the network. In short, the Xception architecture is a linear stack of depthwise separable convolution layers with residual connections. This makes the architecture very easy to define and modify. The above Xception Architecture visualization show above can also be explained thoroughly. The data first goes through the entry flow, then through the middle flow which is repeated eight times, and finally through the exit flow. All Convolution and SeparableConvolution layers are followed by batch normalization (not included in the diagram). All SeparableConvolution layers use a depth multiplier of 1 (no depth expansion).

BU CS440 CLASS CHALLENGE REPORT

Name: Tegveer Ghura

Description of overall architecture used:

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
xception (Functional)        (None, 7, 7, 2048)        20861480
_________________________________________________________________
flatten_4 (Flatten)          (None, 100352)            0
_________________________________________________________________
dropout_4 (Dropout)          (None, 100352)            0
_________________________________________________________________
dense_feature (Dense)        (None, 256)               25690368
_________________________________________________________________
dropout_5 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_4 (Dense)              (None, 4)                 1028
=================================================================
Total params: 46,552,876
Trainable params: 46,498,348
Non-trainable params: 54,528
```

The pre-trained network used is Xception with ImageNet weights for transfer learning. Specifically, for our Xception pre-trained network, we do not include the 3 fully-connected layers at the top of the network ("include_top=False"), but fit the Xception network to a series of densely-connected layers that we include ourselves. The Xception network takes in the input shape of (224,224,3), which corresponds to width and height of 224 and 3 input channels (channels_last data format). The first dense layer is a hidden layer with 256 units of output space and activation function ReLU (f(x)=max(0,x)). The second dense layer is our output layer with 4 output units and a soft-max activation function due to multiclass classification. Before we specify our first dense layer, we initialize a flatten layer, which prepares a vector from image data input for the fully connected layers. We also include two Dropout layers with values of 0.2 each because when I initially tried the Xception Architecture without any Dropout Layers, I experienced extremely high overfitting on the training data. Therefore, including some type of regularization method was helpful to reduce overfitting through model averaging our network.

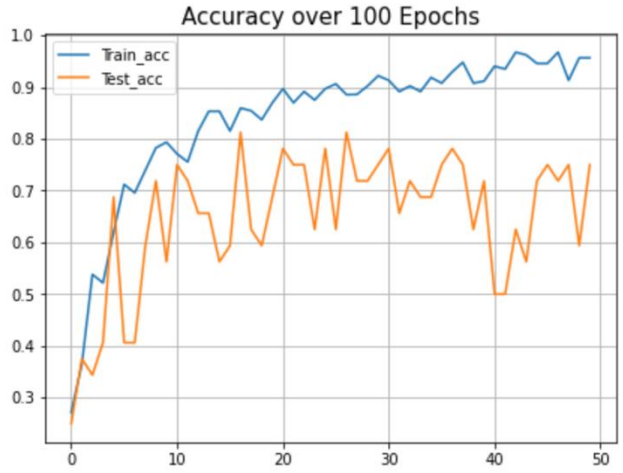
BU CS440 CLASS CHALLENGE REPORT

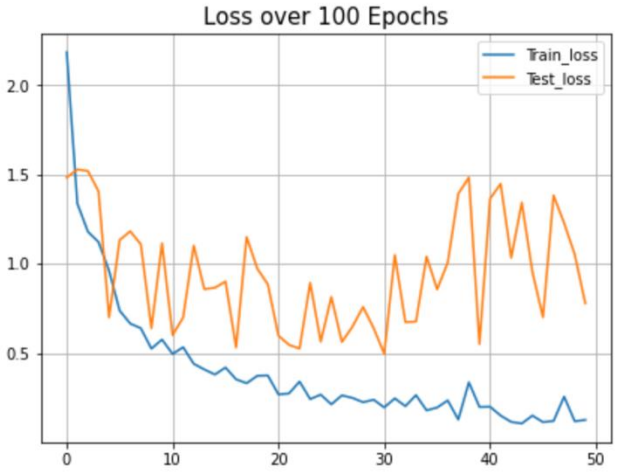
Name: Tegveer Ghura

Optimizer, loss function, parameters, and any regularization used:

The optimizer used is the adaptive, state-of-the-art Adam optimizer that works well in practice and compares favorably to other adaptive learning-method algorithms, as highlighted by its authors. Therefore, Adam computes adaptive learning rates for each parameter and also keeps an exponentially decaying average of past gradients, similar to momentum. The loss function used is categorical cross entropy because our task is that of multiclass classification (4 target classes: Covid-19 vs Normal vs Pneumonia_Bac vs Pneumonia_Vir). This is also a reason why we used the soft-max activation function, instead of sigmoid, in our second dense layer with 4 output units. Parameters used to fit the model include a higher batch size of 32 and 50 training epochs with 6 steps per training epoch or 6 batches per training epoch. The batch size was increased compared to that of Task 1 because the dataset for this task was more abundant and required better computational speed. Moreover, in order to account for the tradeoff between computational speed and convergence of the model, a lower learning rate of 0.0001 was used. The other regularization method used is Data Augmentation by using the ImageDataGenerator() class of Tensorflow. The exact same preprocessing arguments were applied here as they were applied in Task 1.

Plot and comment on the accuracy and the loss:





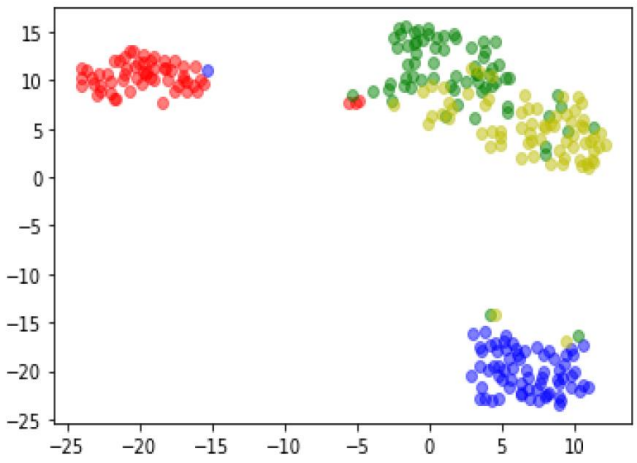

```
36/36 [==============================] - 7s 205ms/step - loss: 0.5290 - accuracy: 0.7778
Test loss: 0.5290159583091736
Test accuracy: 0.7777777910232544
```

BU CS440 CLASS CHALLENGE REPORT

Name: Tegveer Ghura

By initially looking at the plot, one may say that the model was overfitting to an extent, which it may have been, but as much as it originally was without dropout layers. The final test accuracy, however, may paint a different picture as we obtain a good, high accuracy of about 78%. Yes, it should be agreed that the model performance could've been better by hyperparameter tuning and, hence, the increasing gap between the training and test accuracies and losses could've been reduced. This was also one of the reasons why I stuck to only 50 and not a 100 epochs, as that could have also led to severe overfitting of our model, as seen by the increasing gaps (training accuracy increasing and training loss decreasing but validation accuracy and loss plateauing off after the 30th epoch). Therefore, early stopping helped me achieve an even better model performance if instead I kept training my model until a 100 epochs!

Plot and comment on the t-SNE visualization:



The model creates separate, compact clusters for covid-19 and normal types (red and blue), with pneumonia_bac and pneumonia_vir clustered together (green and yellow) but well away from covid-19 and normal. The reason pneumonia_bac and pneumonia_vir are clustered together, but still separated to a great extent within that cluster, is that they both are very different from covid-19 and normal but not very distinct from each other. Therefore, our model does well to distinguish between the 4 classes, given that test accuracy is close to 80%.

BU CS440 CLASS CHALLENGE REPORT

Name: Tegveer Ghura

Detailed Description of ResNet50V2 Architecture:

The major differences between ResNetV1 and ResNetV2:

I. ResNet V1 adds the second non-linearity after the addition operation is performed in between the ***x*** and ***F(x)*.** ResNet V2 has removed the last non-linearity, therefore, clearing the path of the input to output in the form of identity connection.

II. ResNet V2 applies Batch Normalization and ReLU activation to the input before the multiplication with the weight matrix (convolution operation). ResNet V1 performs the convolution followed by Batch Normalization and ReLU activation.

III. The ResNet V2 mainly focuses on making the second non-linearity as an identity mapping i.e. the output of addition operation between the identity mapping and the residual mapping should be passed as it is to the next block for further processing. However, the output of the addition operation in ResNet V1 passes from ReLU activation and then transferred to the next block as the input.

Key Features of ResNet:

I. ResNet uses Batch Normalization at its core. The Batch Normalization adjusts the input layer to increase the performance of the network. The problem of covariate shift is mitigated.

II. ResNet makes use of the Identity Connection, which helps to protect the network from vanishing gradient problem.

III. Deep Residual Network uses bottleneck residual block design to increase the performance of the network.

BU CS440 CLASS CHALLENGE REPORT

Name: Tegveer Ghura

Description of overall architecture used:

```
Model: "sequential_7"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
resnet50v2 (Functional)      (None, 7, 7, 2048)        23564800
_________________________________________________________________
flatten_7 (Flatten)          (None, 100352)            0
_________________________________________________________________
dense_feature (Dense)        (None, 256)               25690368
_________________________________________________________________
dropout_8 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_7 (Dense)              (None, 4)                 1028
=================================================================
Total params: 49,256,196
Trainable params: 49,210,756
Non-trainable params: 45,440
```

The pre-trained network used is ResNet50V2 with ImageNet weights for transfer learning. Specifically, for our ResNet50V2 pre-trained network, we do not include the 3 fully-connected layers at the top of the network ("include_top=False"), but fit the ResNet50V2 network to a series of densely-connected layers that we include ourselves. The ResNet50V2 network takes in the input shape of (224,224,3), which corresponds to width and height of 224 and 3 input channels (channels_last data format). The first dense layer is a hidden layer with 256 units of output space and activation function ReLU (f(x)=max(0,x)). The second dense layer is our output layer with 4 output units and a soft-max activation function due to multiclass classification. Before we specify our first dense layer, we initialize a flatten layer, which prepares a vector from image data input for the fully connected layers. For this model, I only implemented a single Dropout layer with value of 0.25.

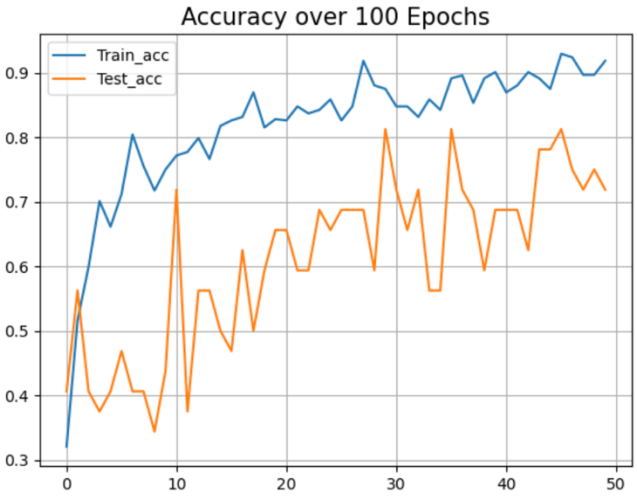
BU CS440 CLASS CHALLENGE REPORT

Name: Tegveer Ghura

Optimizer, loss function, parameters, and any regularization used:

The optimizer used is the adaptive, state-of-the-art Adam optimizer that works well in practice and compares favorably to other adaptive learning-method algorithms, as highlighted by its authors. Therefore, Adam computes adaptive learning rates for each parameter and also keeps an exponentially decaying average of past gradients, similar to momentum. The loss function used is categorical cross entropy because our task is that of multiclass classification (4 target classes: Covid-19 vs Normal vs Pneumonia_Bac vs Pneumonia_Vir). This is also a reason why we used the soft-max activation function, instead of sigmoid, in our second dense layer with 4 output units. All other parameters used in the previous model apply here as well for better comparison of both models (keeping parameters and layers fixed, with mainly only architectures being different).

Plot and comment on the accuracy and the loss:

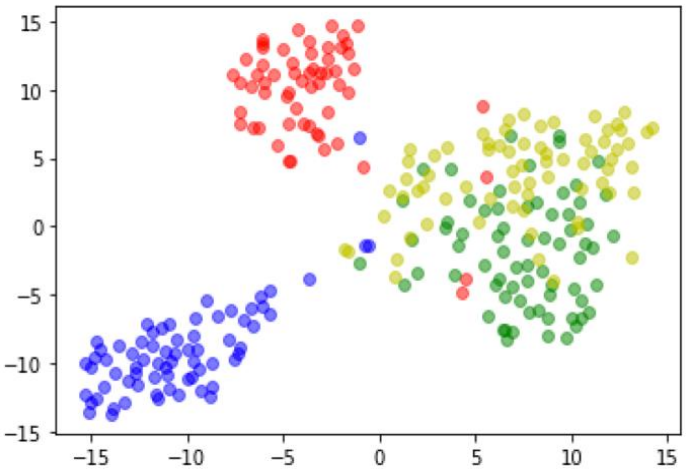



```
36/36 [==============================] - 5s 146ms/step - loss: 2.9377 - accuracy: 0.6111
Test loss: 2.937669277191162
Test accuracy: 0.6111111044883728
```

We notice a similar feature even for this model's plot that there is a good distance between the training and test accuracies and losses. However, the good trend is that the distance between the two lines stays very much constant across all 50 epochs, with both lines following the same trend in both plots. Therefore, chances of overfitting in this architecture are lower. Plus, test accuracy kept increasing and loss kept decreasing even after the 50th epoch, which meant that we could've

trained this model for maybe 75-100 epochs to extract its best performance. The caveat, however, is that overall test accuracy is almost 17% lower and test loss is ~2.5 more for this model than it was for the previous model. Therefore, I believe that if this model were to be trained for more than 50 epochs, it could've reached a test accuracy close to that of 78% or maybe even higher!

Plot and comment on the t-SNE visualization:



This model, although creates separate, compact clusters for covid-19 and normal types (red and blue), the previous model performed better on creating these clusters more further apart. Again, the reason pneumonia_bac and pneumonia_vir are clustered together, but still separated to a great extent within that cluster, is that they both are very different from covid-19 and normal but not very distinct from each other. Moreover the test accuracy of this model is lower than that of the previous Xception model and, hence, we observe that the clusters are not very distinctly separated from each other.