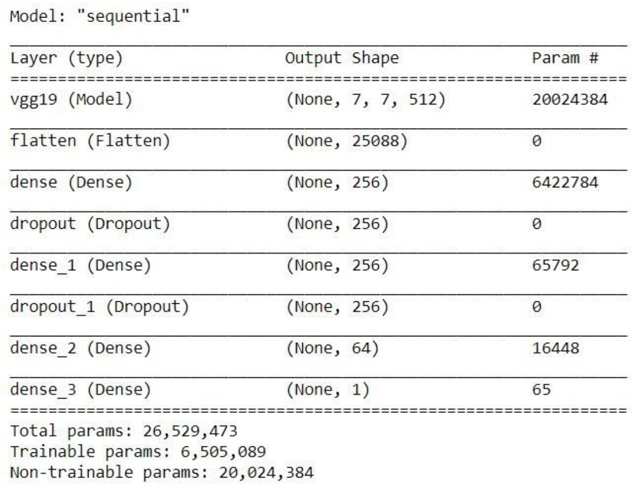
CS 542 Class Challenge: Image Classification of COVID-19 X-rays

Tiam Moradi
BU ID: U15840381
4/19/2020

Task 1

I decided to use VGG19 for this classification and tuned certain hyperparameters in order to achieve the best result I could. VGG19 is a deep neural network that contains 19 weighted layers. This network's input is a (224,224,3) image and begins with convolution layers, with a kernel size of (3x3), and the number of output channels is 64. Then a (2,2) max-pooling layer is followed to reduce the dimensionality. Convolutional layers are applied again, with the same kernel size, but the number of output channels is 128. As you go deeper into the network, the convolutional layers increase from two to four at a time, and the number of output channels increases from 128 to 256, and finally 512. In addition, (2,2) max-pooling is applied after every block of convolutional layers. After the last pooling layer, you flatten the output channels and train three dense layers that contain 4096, 4096, and 1000 nodes. Finally softmax is applied to get predictions.

For this particular task, I kept the VGG19 architecture the same, but I removed pre trained fully connected layers. In my architecture, I added three fully connected layers; two had 256 nodes each , and the third contained 64 nodes. The output layer is a single node with a sigmoid activation. Here is my model's architecture.

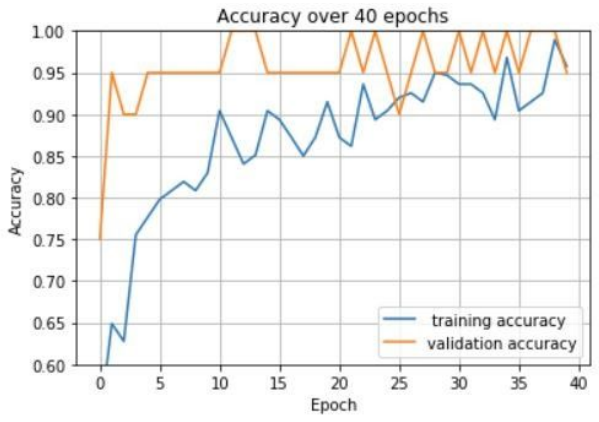


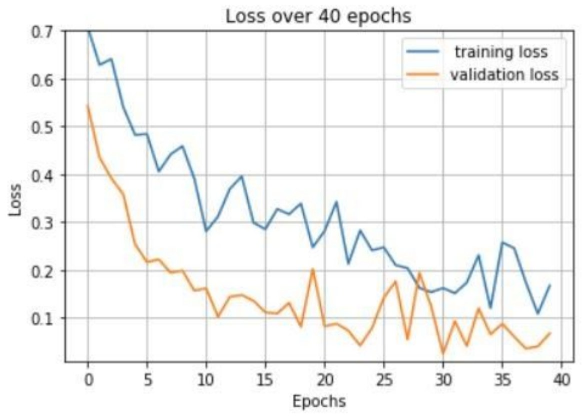
```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg19 (Model)                (None, 7, 7, 512)         20024384
_________________________________________________________________
flatten (Flatten)            (None, 25088)             0
_________________________________________________________________
dense (Dense)                (None, 256)               6422784
_________________________________________________________________
dropout (Dropout)            (None, 256)               0
_________________________________________________________________
dense_1 (Dense)              (None, 256)               65792
_________________________________________________________________
dropout_1 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_2 (Dense)              (None, 64)                16448
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 65
=================================================================
Total params: 26,529,473
Trainable params: 6,505,089
Non-trainable params: 20,024,384
```

I added regularization with Dropout layers. Each Dropout layer zeros out a perceptron with a probability of 25 percent. I tried a variety of probabilities, ranging from 15 percent to 50 percent and found this value to not cause high bias or high variance. I also tried adding Dropout after the flattening layer and before the output layer, however that hindered my performance. I also tried Early Stopping to try to prevent overfitting; I monitored the validation set loss and used a persistence ranging from 3-5. With this method, I always converged earlier than the number of epochs designated, however this did not prevent the model from overfitting and it was often better to not use Early Stopping.

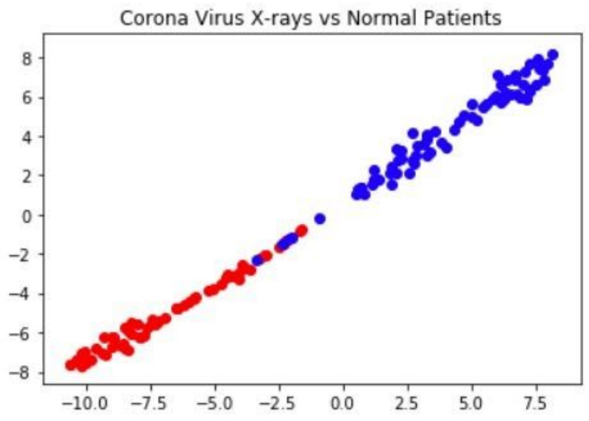
My learning rate was .0001. I tried different values, ranging from .001, to .0005, and the learning rate I chose was not overshooting past a minimum, while not taking too slowly to converge. I also chose to use the number of Epochs to be 40. When I increased the number of epochs over 50, all architectures I tried significantly overfitted. For optimization I used the Adam, and my loss function was Binary_Crossentropy.

Here are the plots of my model's accuracy and loss:





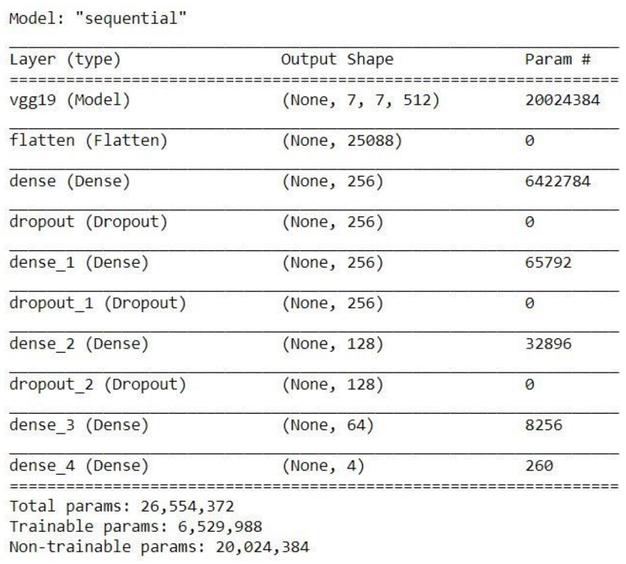
For both the training and validation set accuracies, we can see that they are trending upwards, but there are moments of better or worse performance, especially the training accuracy; this is most likely due to the regularization that is being used. This also explains why the validation accuracy is better than the training accuracy; During training the model does not have access to certain parameters and in validation it is able to use all weights. The loss for the training and validation sets are decreasing, with a few spikes from misclassification. These values are still impressive considering that the loss values are all less than 1, and after the 10th epoch, less than .4. The regularization is also the reason why the validation loss is lower than the training loss; the validation set has access to all weights and perceptrons. To see the models performance on unseen data, I evaluated the model on data from a test directory, and received an 89 percent accuracy; meaning that the model has learned to generalize between the two classes.



Finally, here is a visualization of the features of the test data. The visualization plot was made using T-SNE to reduce the dimensionality of the data into a 2 dimensional feature space. Blue data points are the normal X-rays, and the red data points are the COVID-19 X-rays. These features were evaluated from the input of my model to the "dense_2" From this visualization, we can see besides the few normal X-rays that are overlapping the red cluster, the two classes are separable and also explains the high accuracy of my model; the model was able to create features that would be able to distinguish between the two classes.

Task 2

For this section, I modified my VGG19 model from task1, as well as used an architecture called DenseNet. The modifications that I made with my original VGG19 model were that I added another Dense layer with 128 nodes, followed by a Dropout layer. I also adjusted the Dropout probability to be 20 percent because anything higher than that, and I was performing worse due to a high bias. Finally, the output layer was changed to include 4 perceptrons, one for each class. Instead of a sigmoid activation, for this multi-class task, I used the logits from the output layer and made the loss function CategoricalCrossEntropy, which is an extension of the BinaryCrossEntropy loss. I kept the optimizer the same as before- Adam. I increased the number of epochs from 40 to 75 because the model would need more time to generalize to more than two classes. Here is the modified VGG19 architecture for task 2:

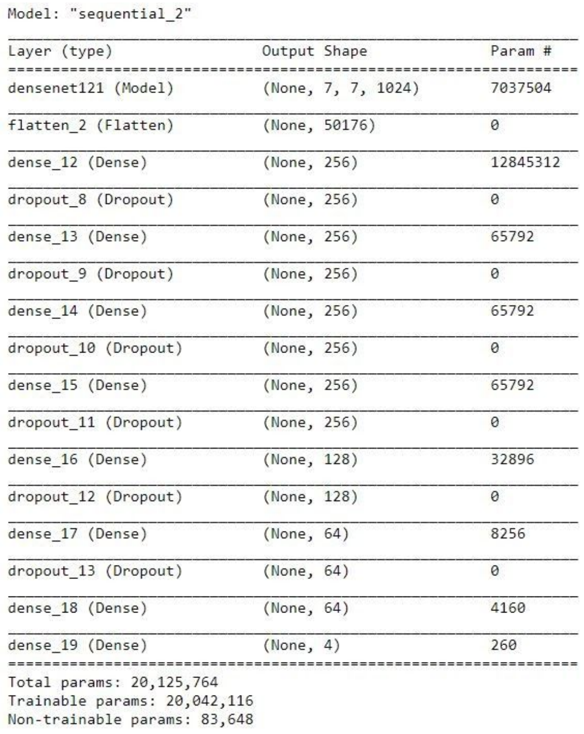


```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg19 (Model)                (None, 7, 7, 512)         20024384
_________________________________________________________________
flatten (Flatten)            (None, 25088)             0
_________________________________________________________________
dense (Dense)                (None, 256)               6422784
_________________________________________________________________
dropout (Dropout)            (None, 256)               0
_________________________________________________________________
dense_1 (Dense)              (None, 256)               65792
_________________________________________________________________
dropout_1 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_2 (Dense)              (None, 128)               32896
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_3 (Dense)              (None, 64)                8256
_________________________________________________________________
dense_4 (Dense)              (None, 4)                 260
=================================================================
Total params: 26,554,372
Trainable params: 6,529,988
Non-trainable params: 20,024,384
```

The second architecture I utilized was DenseNet. This network was designed with the idea that deep networks can deal with the vanishing gradient problem; due to the large number of layers to backpropagate through, the gradient going back to the first layer can be close to zero, thus not updating the parameters and making training significantly longer and lead to worse performance. The architecture of DenseNet has the same Input as the VGG19 architecture, with the image of the X-rays (224,224,3), but the rest of the architecture is very different in comparison.The image goes through a 1D convolution layer, followed by a (2,2) max-pooling layer; this is done in order to preserve spatial resolution while reducing the depth of the image. We then pass the feature map through a Dense block. This block consists of convolutional layers with (3,3) kernels, but the inputs of each convolution layer would be the concatenation of the outputs of the previous layers, including the original input of the dense block.The number of channels increases after each Dense block, going from 128, to 256, to 512, and finally 1024.

Unlike with VGG19, where the number of parameters is quite large with 6.5 million parameters to train, the DenseNet architecture can have either significantly more or less parameters to train. For instance, if we were to do GlobalAveragePooling after the DenseNet model and added fully connected layers to learn our data, then we would only need to train around 300,000 parameters. This is because this particular layer would take the average of each feature map, and return a vector with the depth being the number of channels we have. Without this layer, we could have up to 20 million parameters to train. I chose to not use a GlobalAveragePooling layer. From my testing, only training the 300,000 parameters leads to underfitting and worse test performance. Another big difference between VGG19 and DenseNet is that the Dense blocks concatenate previous output feature maps with original dense block input, while VGG19 only contains input from the immediate previous layer; DenseNet contains more collective knowledge when learning features compared to VGG19.

Here is my modified architecture of the DenseNet model:

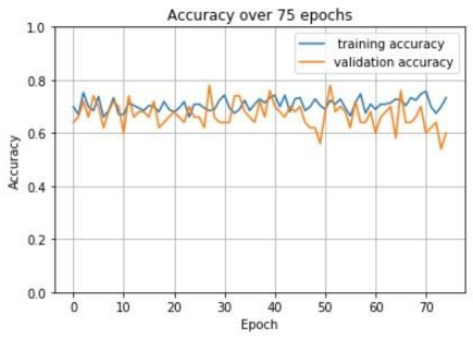


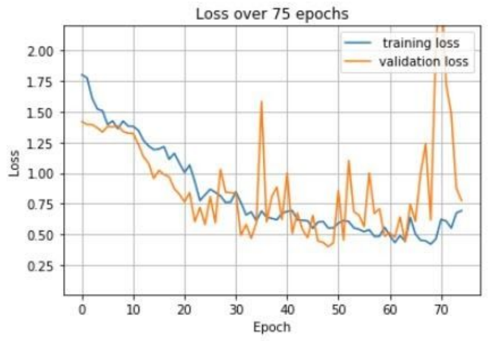
```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
densenet121 (Model)          (None, 7, 7, 1024)        7037504
_________________________________________________________________
flatten_2 (Flatten)          (None, 50176)             0
_________________________________________________________________
dense_12 (Dense)             (None, 256)               12845312
_________________________________________________________________
dropout_8 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_13 (Dense)             (None, 256)               65792
_________________________________________________________________
dropout_9 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_14 (Dense)             (None, 256)               65792
_________________________________________________________________
dropout_10 (Dropout)         (None, 256)               0
_________________________________________________________________
dense_15 (Dense)             (None, 256)               65792
_________________________________________________________________
dropout_11 (Dropout)         (None, 256)               0
_________________________________________________________________
dense_16 (Dense)             (None, 128)               32896
_________________________________________________________________
dropout_12 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_17 (Dense)             (None, 64)                8256
_________________________________________________________________
dropout_13 (Dropout)         (None, 64)                0
_________________________________________________________________
dense_18 (Dense)             (None, 64)                4160
_________________________________________________________________
dense_19 (Dense)             (None, 4)                 260
=================================================================
Total params: 20,125,764
Trainable params: 20,042,116
Non-trainable params: 83,648
```

For this task, I used 75 epochs because if I used higher epochs, I began to overfit to the training and validation set, and perform significantly worse on the test set. When I used too few epochs or used Early Stopping, I would often have a model that would underfit; since the persistence would be satisfied if the validation loss would be similar for a certain amount of epochs, it often not have been exposed to enough training and validation data to be able to generalize to the test set. For both models, I decided to use the learning rate from task 1. Similarly, whenever I tried to raise the learning rate to be greater than .001, I would begin to pass a minimum and see a decrease in accuracy. Setting the learning rate to be less than .0005 would make the model learn too slowly.

For regularization methods, I also included Dropout. In comparison to my VGG19 model, I tried similar ranges of probabilities to zero out certain perceptrons in the fully-connected layers, and anything over .4 would cause the model to underfit; I used a probability of 25 percent. I also used the logits from the output layer and set my loss to CategoricalCrossEntropy with the optimizer being Adam.

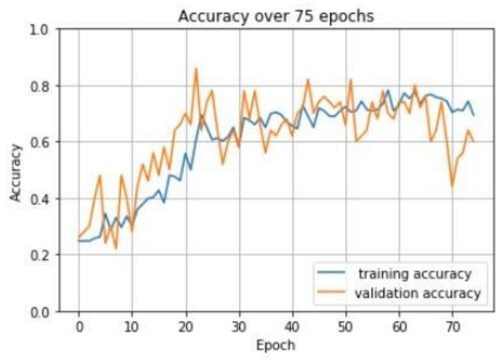
Here is the accuracy and loss of the modified VGG 19 Model.

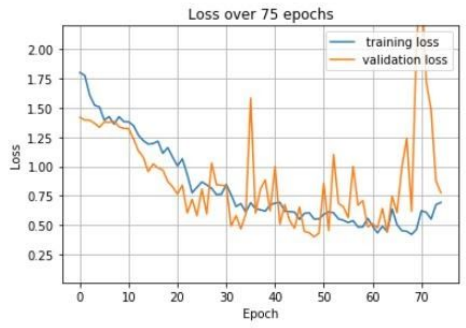




Unlike in task 1 , the training and validation accuracies are really close together, with the training set performing a little bit better. It could be possible that the training batches and the validation batches were of the same distribution between the multiple classes. Around epoch 50 we see that the model starts to overfit to the training set, and that is also evident with the loss graph on the right. For the validation loss, we see huge peakes for misclassification and overall worse performance. To possibly improve the performance, I could have added more layers to the network, which could keep learning new non-linear features to be able to distinguish between the classes.

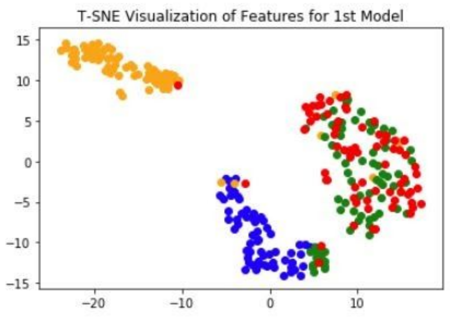
Here is my accuracy and loss for the modified DenseNet model:

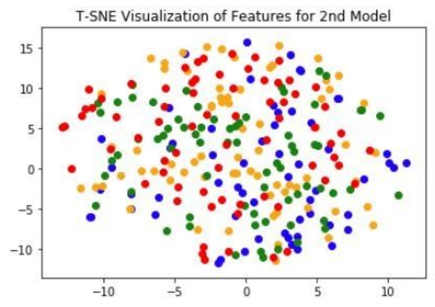




Here we can see that the trajectory of the accuracies of the training and validation improving over the number of epochs. Around the 60th epoch, we can see that the model is starting to overfit to the training data, especially towards the end of training. This can also explain the increase in the loss of the validation set towards the last 10 epochs , with a few spikes earlier. I believe this happened because it was not able to learn features to distinguish between the classes. With this model, possibly adding more regularization could have prevented the later overfitting, possibly add more layers to the model, or increase the number of nodes per fully-connected layer in order to encapsulate more of the previous outputs. The surprising part was that the test accuracy was the same for both models, with the VGG19 model having slightly higher loss. The Test accuracy was 66 percent. This will also be more interesting as we look at the plot of the test data.

Here are visualizations of the features learned by the two models. I also used TSNE to reduce the dimensionality to two components in order to visualize the test data.





In each of the plots, the blue represents COVID-19, orange is normal , green is pneumonia-bacterial, and red is pneumonia-viral. The 1st model is the modified VGG19 model and the 2nd model denotes the modified DenseNet. For the VGG19 visual, I used its input layer and the output was the “dense_2” layer from the architecture. We can see that at this point in the model, it was able to extract features from the test set that was able to clearly separate normal, COVID-19, and Pneumonia X-rays. There was some overlap between Pneumonia bacteria and COVID-19, which has a little gap between the majority of the Pneumonia cases; maybe these X-ray images could represent early signs of Pneumonia and are similar to COVID-19 X-ray. It did have a hard time distinguishing the Pneumonia-bacterial vs Pneumonia-viral. These X-rays could virtually be the same with some minor details that the model was not able to understand. One could consider turning the Pneumonia X-rays into a binary classification problem and try transfer learning for feature extraction.

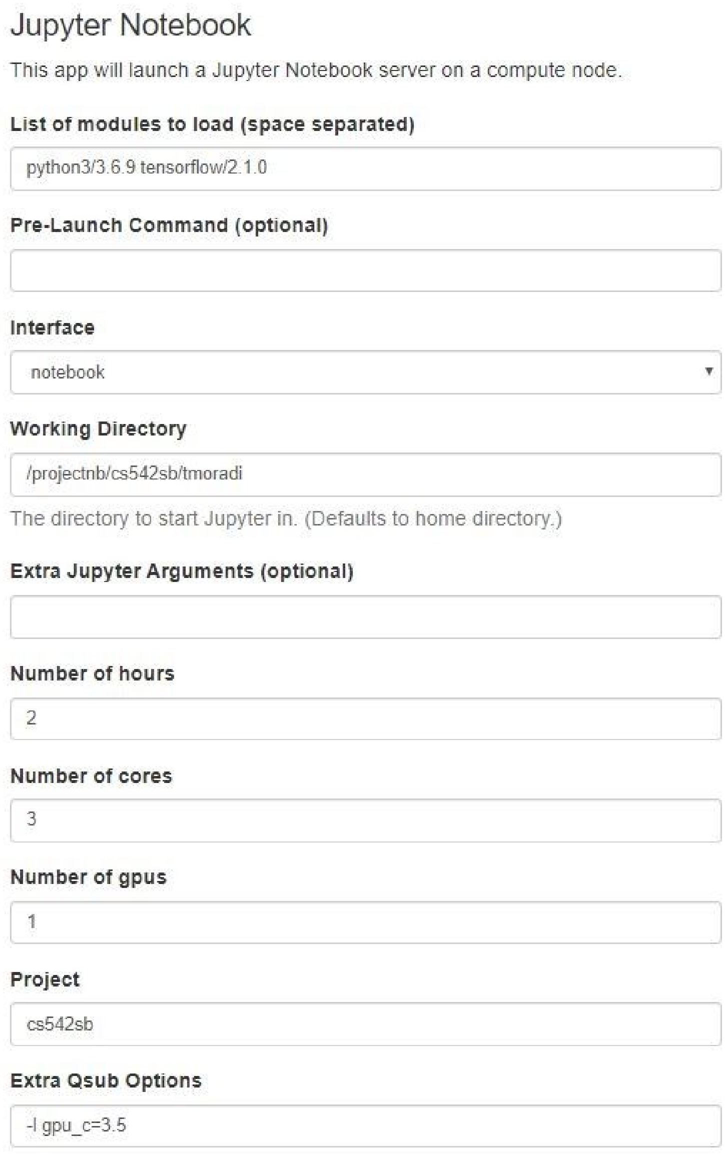
In comparison to the VGG19 model, the DenseNet model looks as though it didn’t capture features that would be able to distinguish between each of the classes. In my opinion, there are two potential reasons why this is the case. First of all, it could be possible that because of the parameter sharing that occurs with the Dense blocks, in that it reuses weights; maybe the parameters that are learned from each of the classes are similar once they are concatenated together.Another reason the clusters may not look as compact is because of the dimensionality reduction. The trade-off with being able to visualize this data is the loss of information. This lost information could have been useful to help the model discriminate between the different classes.

For instance, if we add another component , maybe the T-SNE plot would have each class among a different depth value, making it look separable to some degree.

In comparison to the TSNE visual from task 1, My modified VGG19 model for task 2 was able to learn how to differentiate normal X-rays and normal X-ray better with a bigger distance between the clusters. However, the DenseNet model did not perform better with the features to separate between normal and COVID-19 X-rays.

Extra Credit

Due to batch job errors that I could not figure out how to resolve, I used an interactive session with Jupyter Notebook. Here are the screenshots in the following order: Interactive Session, Devices, GPU time , and CPU time. The times are in terms of minutes.



Jupyter Notebook

This app will launch a Jupyter Notebook server on a compute node.

List of modules to load (space separated)
python3/3.6.9 tensorflow/2.1.0

Pre-Launch Command (optional)

Interface
notebook

Working Directory
/projectnb/cs542sb/tmoradi

The directory to start Jupyter in. (Defaults to home directory.)

Extra Jupyter Arguments (optional)

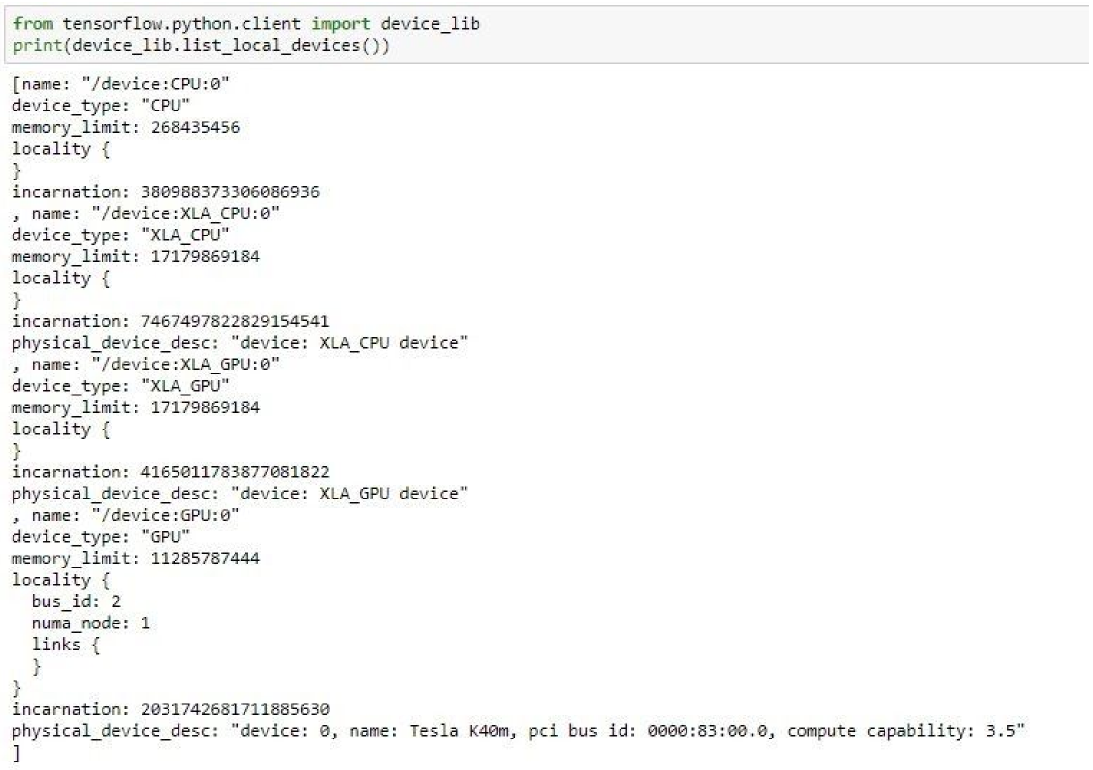
Number of hours
2

Number of cores
3

Number of gpus
1

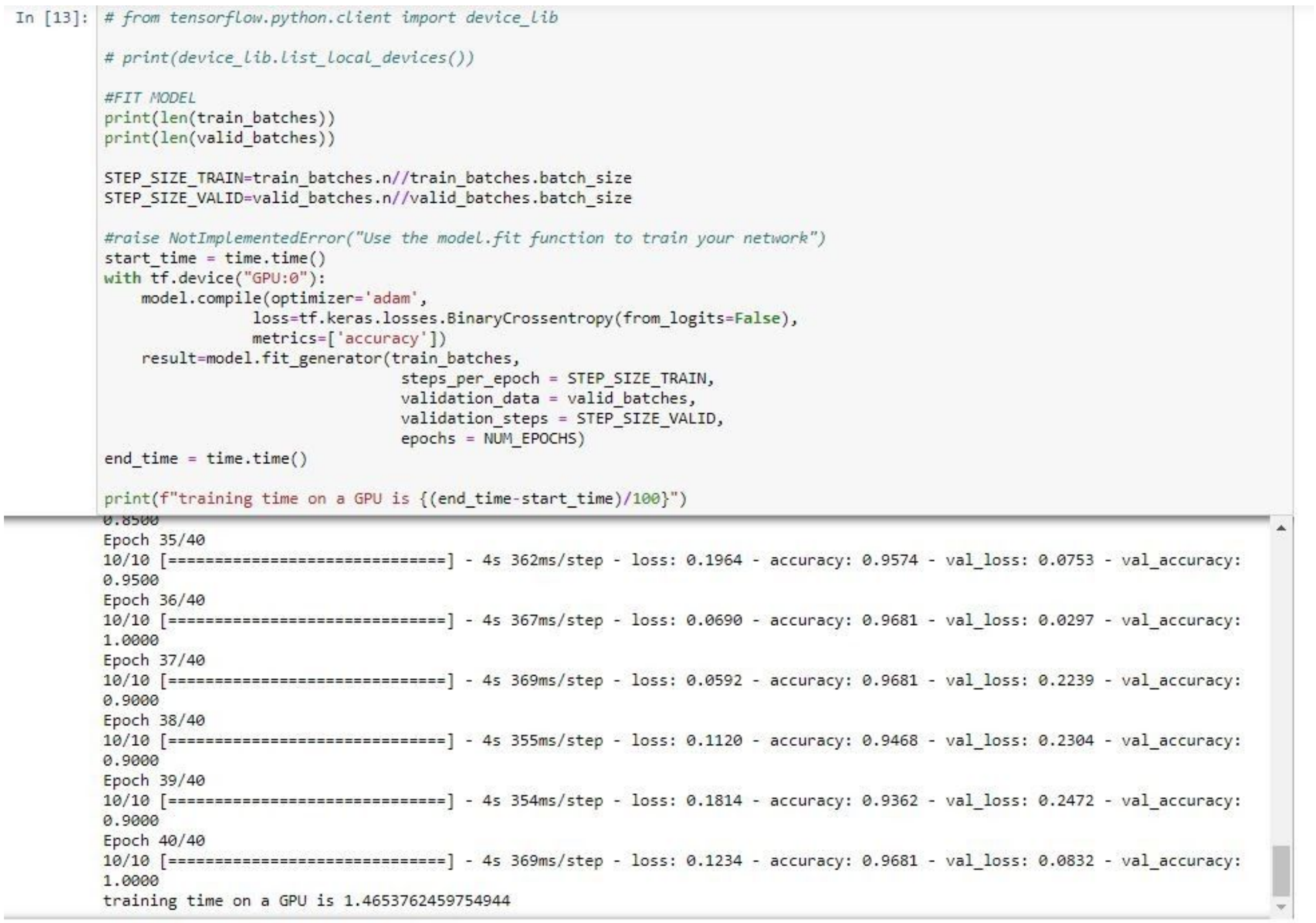
Project
cs542sb

Extra Qsub Options
-l gpu_c=3.5



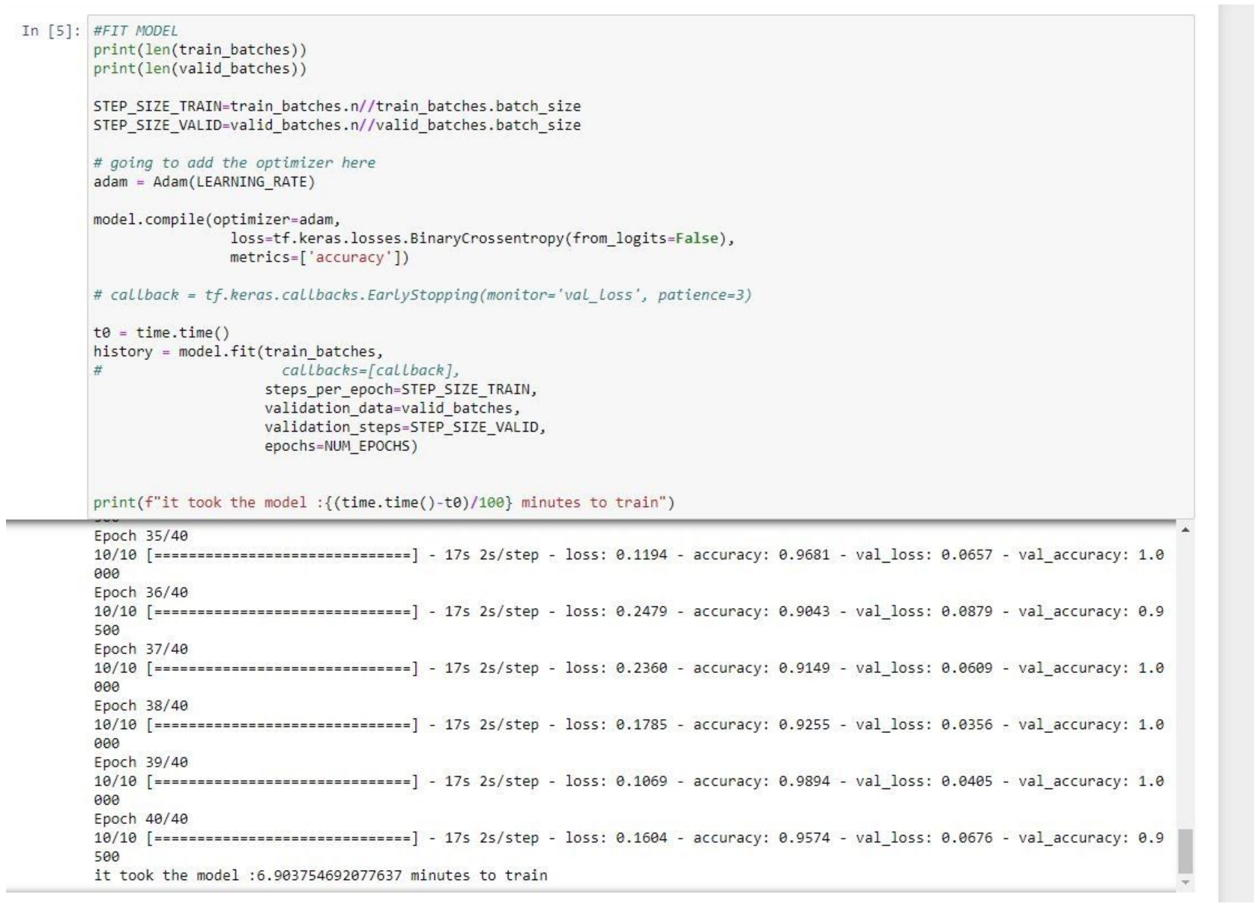
```
from tensorflow.python.client import device_lib
print(device_lib.list_local_devices())
```

```
[name: "/device:CPU:0"
device_type: "CPU"
memory_limit: 268435456
locality {
}
incarnation: 3809883733060869336
, name: "/device:XLA_CPU:0"
device_type: "XLA_CPU"
memory_limit: 17179869184
locality {
}
incarnation: 7467497822829154541
physical_device_desc: "device: XLA_CPU device"
, name: "/device:XLA_GPU:0"
device_type: "XLA_GPU"
memory_limit: 17179869184
locality {
}
incarnation: 4165011783877081822
physical_device_desc: "device: XLA_GPU device"
, name: "/device:GPU:0"
device_type: "GPU"
memory_limit: 11285787444
locality {
  bus_id: 2
  numa_node: 1
  links {
  }
}
incarnation: 2031742681711885630
physical_device_desc: "device: 0, name: Tesla K40m, pci bus id: 0000:83:00.0, compute capability: 3.5"
]
```



```
In [13]: # from tensorflow.python.client import device_lib

         # print(device_lib.list_local_devices())

         #FIT MODEL
         print(len(train_batches))
         print(len(valid_batches))

         STEP_SIZE_TRAIN=train_batches.n//train_batches.batch_size
         STEP_SIZE_VALID=valid_batches.n//valid_batches.batch_size

         #raise NotImplementedError("Use the model.fit function to train your network")
         start_time = time.time()
         with tf.device("GPU:0"):
             model.compile(optimizer='adam',
                         loss=tf.keras.losses.BinaryCrossentropy(from_logits=False),
                         metrics=['accuracy'])
             result=model.fit_generator(train_batches,
                                         steps_per_epoch = STEP_SIZE_TRAIN,
                                         validation_data = valid_batches,
                                         validation_steps = STEP_SIZE_VALID,
                                         epochs = NUM_EPOCHS)
         end_time = time.time()

         print(f"training time on a GPU is {(end_time-start_time)/100}")
```

```
0.8500
Epoch 35/40
10/10 [==============================] - 4s 362ms/step - loss: 0.1964 - accuracy: 0.9574 - val_loss: 0.0753 - val_accuracy:
0.9500
Epoch 36/40
10/10 [==============================] - 4s 367ms/step - loss: 0.0690 - accuracy: 0.9681 - val_loss: 0.0297 - val_accuracy:
1.0000
Epoch 37/40
10/10 [==============================] - 4s 369ms/step - loss: 0.0592 - accuracy: 0.9681 - val_loss: 0.2239 - val_accuracy:
0.9000
Epoch 38/40
10/10 [==============================] - 4s 355ms/step - loss: 0.1120 - accuracy: 0.9468 - val_loss: 0.2304 - val_accuracy:
0.9000
Epoch 39/40
10/10 [==============================] - 4s 354ms/step - loss: 0.1814 - accuracy: 0.9362 - val_loss: 0.2472 - val_accuracy:
0.9000
Epoch 40/40
10/10 [==============================] - 4s 369ms/step - loss: 0.1234 - accuracy: 0.9681 - val_loss: 0.0832 - val_accuracy:
1.0000
training time on a GPU is 1.4653762459754944
```



```
In [5]: #FIT MODEL
        print(len(train_batches))
        print(len(valid_batches))

        STEP_SIZE_TRAIN=train_batches.n//train_batches.batch_size
        STEP_SIZE_VALID=valid_batches.n//valid_batches.batch_size

        # going to add the optimizer here
        adam = Adam(LEARNING_RATE)

        model.compile(optimizer=adam,
                      loss=tf.keras.losses.BinaryCrossentropy(from_logits=False),
                      metrics=['accuracy'])

        # callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=3)

        t0 = time.time()
        history = model.fit(train_batches,
        #                     callbacks=[callback],
                            steps_per_epoch=STEP_SIZE_TRAIN,
                            validation_data=valid_batches,
                            validation_steps=STEP_SIZE_VALID,
                            epochs=NUM_EPOCHS)


        print(f"it took the model :{(time.time()-t0)/100} minutes to train")
```

```
Epoch 35/40
10/10 [==============================] - 17s 2s/step - loss: 0.1194 - accuracy: 0.9681 - val_loss: 0.0657 - val_accuracy: 1.0
000
Epoch 36/40
10/10 [==============================] - 17s 2s/step - loss: 0.2479 - accuracy: 0.9043 - val_loss: 0.0879 - val_accuracy: 0.9
500
Epoch 37/40
10/10 [==============================] - 17s 2s/step - loss: 0.2360 - accuracy: 0.9149 - val_loss: 0.0609 - val_accuracy: 1.0
000
Epoch 38/40
10/10 [==============================] - 17s 2s/step - loss: 0.1785 - accuracy: 0.9255 - val_loss: 0.0356 - val_accuracy: 1.0
000
Epoch 39/40
10/10 [==============================] - 17s 2s/step - loss: 0.1069 - accuracy: 0.9894 - val_loss: 0.0405 - val_accuracy: 1.0
000
Epoch 40/40
10/10 [==============================] - 17s 2s/step - loss: 0.1604 - accuracy: 0.9574 - val_loss: 0.0676 - val_accuracy: 0.9
500
it took the model :6.903754692077637 minutes to train
```