Machine Learning - Kaggle Final Project

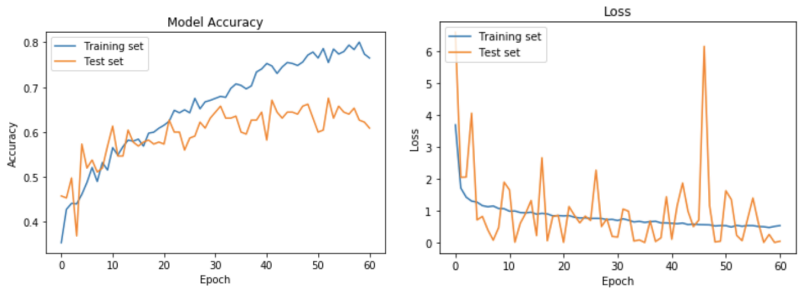
Julie Song js5360

May 9, 2020

Writeup

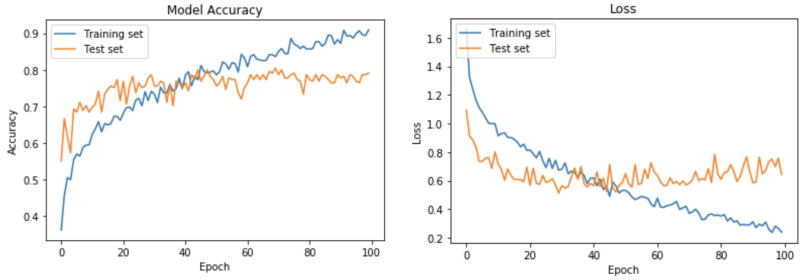
1. Methods/algorithms

I approached the problem with a convolutional neural network approach, eventually using transfer learning with a pretrained VGG model. Prior to starting the code on the assignment, I looked into the best CNN models for image classification using this website: https://machinelearningmastery.com/review-of-architectural-innovations-for-convolutional-neural-networks-for-image-classification/ and decided it would allow for simple and direct use of CNNs.
The difference in approaches really came down to whether or not I used transfer learning. My first approach did involve importing the VGG model from keras.applications but it failed to create weights and features that were essential to transfer learning. As a result, it wasn't much different from implementing a VGG model from scratch, which takes an immense amount of time to run. For this, each epoch took about 90-120 seconds to run and needed an average of around 50 epochs for the validation loss to converge. Due to such drawbacks in time, I could only use up to the second block of the base VGG model, which I feared would be insufficient. As a result, my reason for switching methodology were not only the horrible accuracies as will be explained later, but also its lack in efficiency. Below are my accuracy and loss functions for this attempt:



You can see how the model is not only overfitting, but is failing to achieve

validation accuracies over 70% with extreme fluctuations. My second approach involved the creation of a bottleneck file system that would convert all pixel images to numpy arrays and store it in the system. I was inspired by this article in particular: https://towardsdatascience.com/a-simple-cnn-multi-image-classifier-31c463324fa. The only remotely time-consuming aspect of this was saving the weights/features of the model in bottleneck files, which still only took a maximum of 3 minutes and 20 seconds across train, validation, and test generators. Afterwards, the process of fitting was extremely efficient, where each epoch took less than a second to run. The decision to use the VGG model shined here especially, considering that it uses only 11 layers and is easy to work with. Below are accuracy and loss functions for this attempt:



Unfortunately, while the accuracies are much better and stable, the model still shows overfitting in how validation accuracies plateau around the 0.80 mark. I will elaborate on this in a later section.

2. Preprocessing

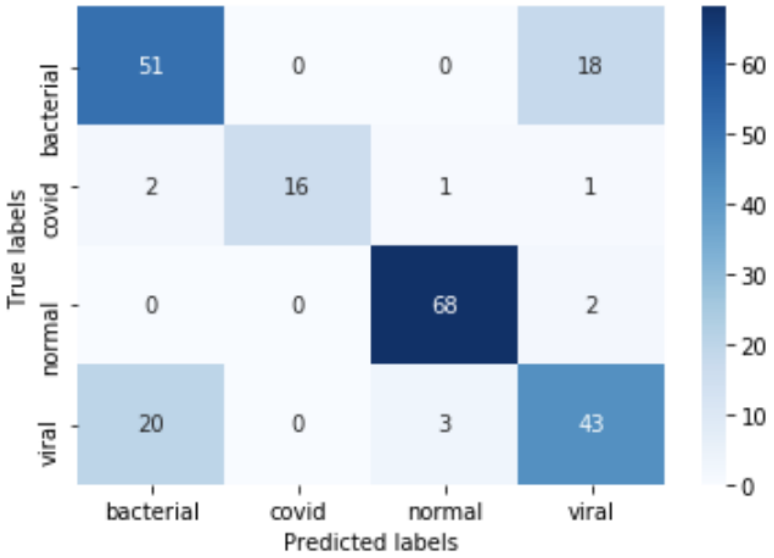
Preprocessing mainly involved splitting the train dataset into train and validation tests as well as using the labels in the train.csv file to organize the images into their respective class folders. I used a 80-20 validation split, which split train and validation sets into 902 and 225 samples, respectively. I fed these directories into generators and normalized the input using rescale=1./255 and preprocessing_input=preprocess_input from keras.applications.vgg16. Though I did randomly choose files from the train folder to put in the validation dataset, I realize in retrospect that it would have been smarter to test my model across random splits each time. In order to do this, I probably would have had to have the data flow from a dataframe instead of a directory. What I had done would have been detrimental considering how I used the same validation test set each time, especially so if the sample distribution is not representative of the actual test set.

3. Error Analysis

The classification metrics across the four classes is as follows:

	precision	recall	f1-score	support
bacterial	0.72	0.72	0.72	69
covid	1.00	0.80	0.89	20
normal	0.94	0.97	0.96	70
viral	0.67	0.65	0.66	66
micro avg	0.80	0.79	0.79	225
macro avg	0.84	0.79	0.81	225
weighted avg	0.80	0.79	0.79	225
samples avg	0.79	0.79	0.79	225

We can see that the f1-scores are highest for normal and covid while relatively low for bacterial and viral. The confusion matrix gave some insight into why that might be the case:



Immediately, we can observe that though the majority of labels are predicted correctly, the model seems to confuse bacterial images with viral images and vice versa. It is also notable how all images with the true label of covid were predicted correctly, thus corresponding with its perfect precision shown in the metrics above.

The severity of these errors is dependent on the situation. If the prime purpose of this model is to distinguish covid images from non-covid images, the confusion between bacterial and viral images would not be of high concern. If this model were to be used in a clinical setting where

identifying covid is the main concern, while such confusion would not be lethal, there would be possibilities of false positives. On the contrary, the fact that the probability of false negatives is very slim makes it a favorable model in discerning covid from other cases.

4. Issues Encountered + Moving On I've encountered many issues during the course of this project. The first few attempts I reached horrendous accuracies in the 20s. This was a stupid mistake on my part since I had accidentally set shuffle=True for the test generator, thus losing track of the filenames and their coresponding labels. Even after fixing that, however, my accuracies were still low in the 50s.

   On top of saving weights/features to a bottleneck system, a few other improvements I had made related to learning rates, optimization functions, normalization, and activation functions. Whereas I previously didn't think a learning rate of 0.01 to be high, I was able to learn that neural networks are highly sensitive to noise in data and thus should be run with very small α values. I had also experimented with various optimization functions, starting with Adam, but read later that Adam was not very compatible with VGG. Eventually, I had come to settle on RMSprop. In regards to normalization, I had realized during a run that my accuracy values were not changing and my loss values were abnormally high. I had previously thought that preprocess_input from keras.preprocessing rescaled for you, but I realized that it was still necessary to set rescale=1./255. Changing ReLU activation to Leaky ReLU improved accuracies as well as it resolves the "dying ReLU" problem and speeds up training since it would no longer have zero-slope parts.

   Going on, one problem that persisted was the problem of overfitting. In attempt to fix this problem, I tried various different methods including data augmentation, drop outs, lessening architecture complexity, and early stopping. Though they did help in making miniscule improvements, the validation accuracy still ended up plateauing at around the 0.80 mark. If I were to go back, I would approach this problem differently in two main ways. First, as mentioned previously, I would try making random validation splits each time in case the innate problem lied with my holdout validation set. Second, I could try altering the default settings of the pretrained VGG weights. I had thought it unnecessary considering the complexity of the problem, but perhaps if we were to tailor the weights/features to the structure of the problem at hand, we'd be able to surpass the 0.80 limitation.