Hyper-parameters Exploration for Mask Usage Detection.

Deep Learning

TC3067.1 Group 1

Aug. – Dic. 2020

Instructor:

Dr. Leonardo Garrido

Greg Espinosa Carrillo A01196764

Mario Manuel Roque Chairez A01282301

November 23, 2020

Introduction

We had a lot of dataset options from different websites but we chose the face mask detection dataset from Kaggle to implement our deep dense neural network. We chose this dataset because right now all the world is suffering from the covid pandemic and one of the best prevention methods is to wear a face mask. The model developed in this work is a binary classifier of human faces images, the purpose of it is to tell if a person is using a mask or not.

According to Rick Kushman, Writer in Public Affairs of the UC Davis Health, the fundamental methods to prevent spreading of COVID-19 are: maintaining social distance, wearing a mask and avoiding outdoor social interactions whenever possible. All these methods reduce the risk of getting infected but many people still ignore them causing more spreadings. The virus can be transmitted through droplets and aerosol particles which come out from the mouth. Many stores and malls implemented obligatory face mask wearing, which is great because wearing the face mask decreases the transmission risk by 65 percent, but there are still individuals who remove their face mask when leaving the checking stations, this is why a NN comes handly by watching the individuals on a vigilance system to alert the guards of people breaking the rules.

In this report we are going to present the data set characteristics, the methodology to obtain the best model through hyper-parameterization and the results of the trained model.

Data Set

The dataset ”Face Mask Detection” was obtained in Kaggle and contains 853 images of people with masks, without masks and with masks worn incorrectly, it also also contains several .xml files with information of every image for example: name of the image, class label, size and the coordinates of the persons faces. The original images were of different sizes, different orientations and with more than 1 person; to make it a binary problem and to standardize the input we preprocessed the dataset.

For the preprocessing stage we first read all the xmls files and obtained the information of all images. Second, only the images with labels “with_mask” and “without_mask” were read and cropped to obtain the face of the main person in the image (this was done using the face coordinates from the xmls files). Third and last, the cropped images were scaled to have a standard size of 64 pixels x 64 pixels. The code of this preprocessing is in the file DatasetInfo.py created by us and is loaded to the Jupyter notebook that uses its methods to load the images and their labels.

In the Jupyter notebook, the images and labels are splitted into 3 sets: train set, dev set and test set. The train set is used to train the model, the dev set to get low bias and low variance through hyper-parametrization, and the test set to evaluate the model as if it was released into production. Here are the sizes of each set.

- Train set: 312 images (70% of all data)
- Dev set: 67 images (15% of all data)
- Test set: 68 images (15% of all data)

Here are some examples of the preprocessed images from the 2 classes: “with_mask” and “without_mask”.

Image 1 - Face with mask

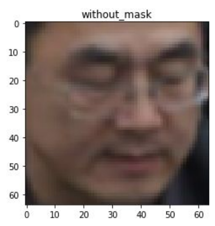


Image 2 - Face without mask

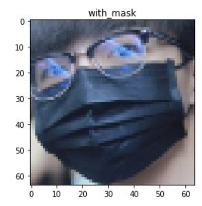


Image 3 - Face with mask

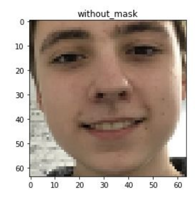


Image 4 - Face without mask

Methodology

First, we had the idea of trying several configurations for the model by hand but we immediately changed to a more automatic approach. The suggested hyper-parameters were: learning rate, number of iterations, number of layers and the number of neurons in each layer.

In the Jupyter notebook we created the helper recursive function rec to have different architectures of the model (different number of layers and neurons). This function generates a list with all the possible combinations given a minimum value, maximum value, max length and current length. Here is an example of the function.

Parameters:

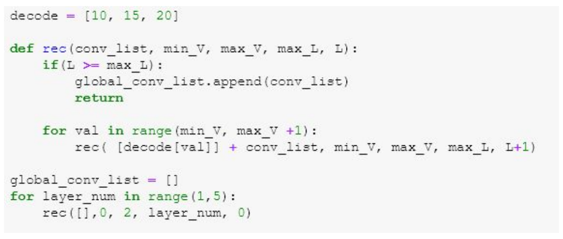
- Minimum value: 0
- Maximum value: 2
- Maximum length: 3
- Current length: 0

Output List (Total combinations: 27):

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

We wanted to try many architectures for the model so we generated all the combinations with a min value of 10, max_value of 15 and max length of 6 but the number of combinations was too high (approximately 46,656). Adding the variation in iterations and learning rate the training time was too high (more than 2 days) so we needed to reduce the combinations.

To reduce the combinations the rec function was modified to insert a number of this list [10, 15, 20] as the value using the current value as the index. So the new parameters were: min_value = 0, max_value = 2, max_length = layer_num and current_lenght = 0 where the layer_num is 1, 2, 3 and 4 and the min_value and max_value become index limits for the list [10, 15, 20]. All this means that we want all the possible combinations from a network that can have 1, 2, 3 or 4 layers, and in those layers it can have 10, 15 or 20 neurons; the total combinations are 120. The function is shown below.



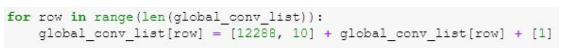
```python
decode = [10, 15, 20]

def rec(conv_list, min_V, max_V, max_L, L):
    if(L >= max_L):
        global_conv_list.append(conv_list)
        return

    for val in range(min_V, max_V +1):
        rec( [decode[val]] + conv_list, min_V, max_V, max_L, L+1)

global_conv_list = []
for layer_num in range(1,5):
    rec([],0, 2, layer_num, 0)
```

After we have all the layer combinations in global_conv_list we proceed to add the input layer, the output layer and an extra static hidden layer of 10 neurons at the beginning. This is done in the following code snippet where the input layer is of size 12288 and the output layer has only 1 neuron.



```python
for row in range(len(global_conv_list)):
    global_conv_list[row] = [12288, 10] + global_conv_list[row] + [1]
```

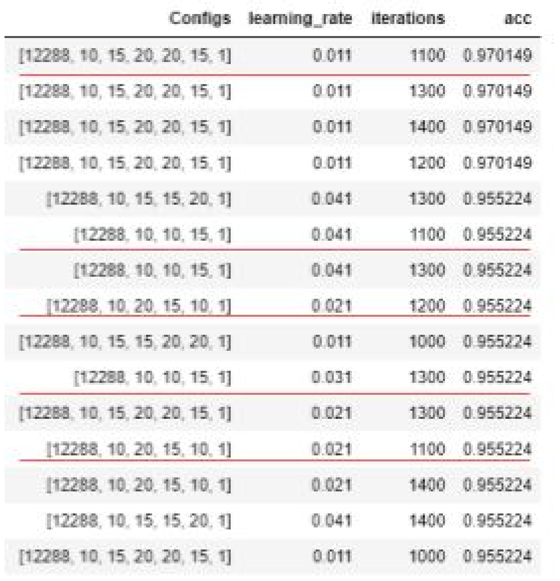
The model was trained automatically in 3 nested for loops. The first for loop varies the learning rate from 0.001 to 0.051 in steps of 0.01. The second loop varies the iterations from 1000 to 1500 in steps of 100. The third loop tries all the 120 layers and neurons combinations. Inside the loops we train with the train set and evaluate the model with the dev set; the accuracy and model configuration were saved in a list so at the end we could look for the best combination. In total, the algorithm requires 3,000 trainings and evaluations to complete all the different variations of parameters. When it finishes, the saved values are passed to a .csv file to preserve the information even if the jupyter notebook is reseted.

From the 3,000 configurations we selected the top 5 results to select the best hidden layer activation function. We tested (using the dev set) the top 5 configurations with the hidden layer activation function ReLu and Tanh, which gave us 10 results. From these results we selected the final and best configuration.

At last with the final and best configuration we trained a model with the training set and evaluated it with the dev and test set.

Results

Table 1 shows the top 15 configurations out of the 3,000 tries.



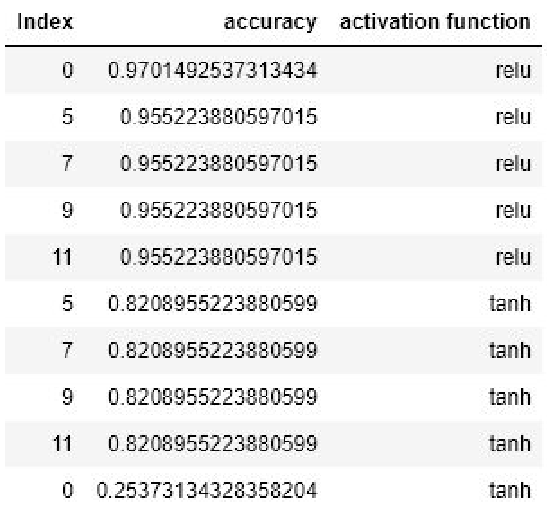
Configs	learning_rate	iterations	acc
[12288, 10, 15, 20, 20, 15, 1]	0.011	1100	0.970149
[12288, 10, 15, 20, 20, 15, 1]	0.011	1300	0.970149
[12288, 10, 15, 20, 20, 15, 1]	0.011	1400	0.970149
[12288, 10, 15, 20, 20, 15, 1]	0.011	1200	0.970149
[12288, 10, 15, 15, 20, 1]	0.041	1300	0.955224
[12288, 10, 10, 15, 1]	0.041	1100	0.955224
[12288, 10, 10, 15, 1]	0.041	1300	0.955224
[12288, 10, 20, 15, 10, 1]	0.021	1200	0.955224
[12288, 10, 15, 15, 20, 20, 1]	0.011	1000	0.955224
[12288, 10, 10, 15, 1]	0.031	1300	0.955224
[12288, 10, 15, 20, 20, 15, 1]	0.021	1300	0.955224
[12288, 10, 20, 15, 10, 1]	0.021	1100	0.955224
[12288, 10, 20, 15, 10, 1]	0.021	1400	0.955224
[12288, 10, 15, 15, 20, 1]	0.041	1400	0.955224
[12288, 10, 15, 20, 20, 15, 1]	0.011	1000	0.955224

Table 1 - Top 15 configurations of 3,000

In table 1 the top 1, 2, 3 and 4 configurations have the same number of layers and neurons but with a different number of iterations, so they are basically the same model. The real top 5 models are underlined in red and were selected based on the following criteria:

- More accuracy
- Less number of layer
- Less number of iterations

As we mentioned in methodology, the top 5 configurations were tested with 2 different hidden layer activation functions. The results of this test were made with the dev set and are shown in Table 2.



Index	accuracy	activation function
0	0.9701492537313434	relu
5	0.955223880597015	relu
7	0.955223880597015	relu
9	0.955223880597015	relu
11	0.955223880597015	relu
5	0.8208955223880599	tanh
7	0.8208955223880599	tanh
9	0.8208955223880599	tanh
11	0.8208955223880599	tanh
0	0.25373134328358204	tanh

Table 2 - Results of top 5 configurations using relu and tanh

As shown in Table 2, the final and best configuration was the top 1 configuration from table 1.

Final results

We created the final model with the best and final configuration and evaluated for the dev and test set. The cost graph is shown in image 5.

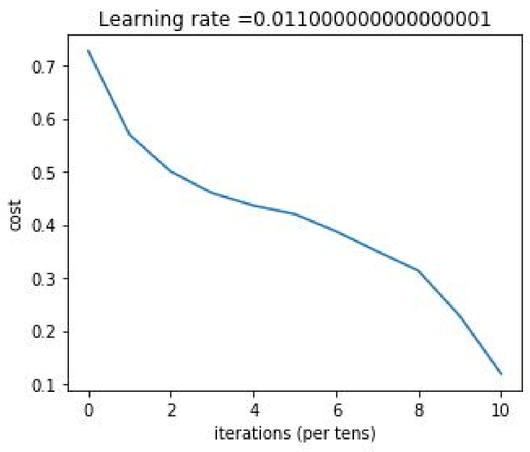


Image 5 - Cost Graph of the best and final configuration

The obtained accuracy for the **dev set was 97.01%** and for the **test set 95.59%**. The mislabeled images of the dev set are shown in image 6 and the ones of test set in image 7.

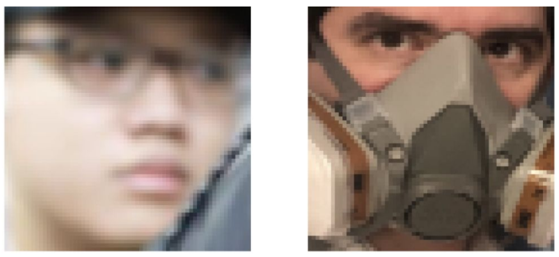


Image 6 - Dev set Mislabeled images

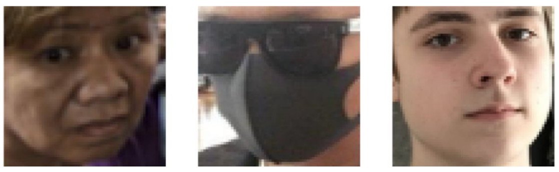


Image 7 - Test set Mislabeled images

A few types of images the model tends to do poorly on include:

- Low resolution images
- Unconventional face masks
- Irregular or low brightness on a picture.

Conclusions

We achieved the objective of this work and obtained acceptable results because our proposed model had high accuracy and low training time (approximately 20 to 30 seconds). Thanks to the seeded randomness our results are reproducible. The functionality of the model is quite simple but with further work it can be integrated into a more complex project. A possible update could be to use a convolutional neural network instead of a deep dense neural network, because CNNs have better performance in image classification.

References

Kushman, R. (2020, July 6). Your Mask Cuts Own Risk by 65 Percent. Retrieved from https://www.ucdavis.edu/coronavirus/news/your-mask-cuts-own-risk-65-percent/

N.A. (20/5/2020). *face-mask-detector*. Retrived from https://www.kaggle.com/notadithyabhat/face-mask-detector

Larxel. (4/5/2020). *Face Mask Detection*, Version 1. Retrieved from https://www.kaggle.com/andrewmvd/face-mask-detection