

Covid-19 severity prediction

ML phase V report

Moneer Zaki
Computer Engineering
The American University in
Cairo
moneerzaki@aucegypt.edu

Mario Ghaly
Computer Engineering
The American University in
Cairo
mariomamdouh@aucegypt.edu

ABSTRACT

The final phase of this machine learning project includes the implementation of both the final model and utility application. Following up on the last phase -- where the model design and application design were planned using fine-tuning plans and diagrams -- in this report, the actual steps done for fine-tuning the hyperparameters of the Tensorflow Neural Network model is going to be demonstrated together with the final application with its APIs, simple preprocessing required and retraining explained.

IMPLEMENTATION CHOICES

Setup

Briefly, for the purpose of making a real-life useful application, the column representing whether the patient is dead or still alive was removed from the dataset in both training and prediction. Then, all hyperparameters experimented were evaluated using 5-fold cross-validation as in milestone 3, but with a small modification. Folds previously had random percentages of positive and negative classes, which passively affected the results. This time the dataset was sampled to make sure that the distribution of the positive class remains balanced across all folds.

Loss Function

The first target was to handle class imbalance as mentioned before (39% vs 61%), so after research, it was found that Binary Focal Cross Entropy Loss function was a great choice for the case at hand. Thus, the first setting was just basic and logical hyperparameters. They are going to be fine-tuned after

the loss function. The hyperparameters at first were as follows:

- **1** hidden layer
- **32** neurons
- **Sigmoid** as the activation function for all perceptrons
- **Learning Rate** = $1e-3$
- **Optimizer** = Adam
- **Early Stopping:**
 - Monitored value = F1-score
 - Patience = 3 (how many worse epochs in F1-score the model will wait before stopping)
 - Restore the best weights in case of early stopping
- **Epochs** = 10
- **Batch Size** = 2048
- **Loss Function** = Focal Loss with its default parameters

Results → No improvements were achieved. However, the evaluation metrics on training data were converging in a good manner, indicating that the learning rate and epochs values are efficient enough, so their initial assignments do not need tuning as shown in the screenshot below. It illustrates how the values of different metrics converge in 10 epochs.

```
accuracy: 0.6323 - auc: 0.6058 - f1_score: 0.5438 - loss: 0.1627 - precision: 0.4842 - recall: 0.0659
accuracy: 0.6640 - auc: 0.6377 - f1_score: 0.5435 - loss: 0.1586 - precision: 0.6262 - recall: 0.2474
accuracy: 0.6653 - auc: 0.6399 - f1_score: 0.5436 - loss: 0.1582 - precision: 0.6265 - recall: 0.2564
accuracy: 0.6652 - auc: 0.6392 - f1_score: 0.5445 - loss: 0.1583 - precision: 0.6278 - recall: 0.2577
accuracy: 0.6657 - auc: 0.6401 - f1_score: 0.5437 - loss: 0.1582 - precision: 0.6267 - recall: 0.2591
accuracy: 0.6658 - auc: 0.6399 - f1_score: 0.5445 - loss: 0.1582 - precision: 0.6301 - recall: 0.2582
accuracy: 0.6659 - auc: 0.6405 - f1_score: 0.5435 - loss: 0.1580 - precision: 0.6277 - recall: 0.2571
accuracy: 0.6654 - auc: 0.6406 - f1_score: 0.5449 - loss: 0.1582 - precision: 0.6302 - recall: 0.2573
accuracy: 0.6665 - auc: 0.6409 - f1_score: 0.5439 - loss: 0.1580 - precision: 0.6306 - recall: 0.2598
accuracy: 0.6673 - auc: 0.6424 - f1_score: 0.5436 - loss: 0.1578 - precision: 0.6326 - recall: 0.2591
```

To try and rapid the learning curve at the first epochs, an output bias was added in the output layer. 2 output biases were tried: one biasing the model towards the minority class and the other towards the majority class. The equation used is:

$$bias = \ln(count_{pos_label} / count_{neg_label}) \quad \text{and} \quad \text{vice-versa}$$

Results → Better metrics at the first epoch were recorded when biasing the model towards the minority class as shown below in comparison with the last screenshot -- in recall, accuracy, f1, and auc.

```
accuracy: 0.6577 - auc: 0.6348 - f1_score: 0.5444 - loss: 0.1624 - precision: 0.6268 - recall: 0.2092
accuracy: 0.6636 - auc: 0.6382 - f1_score: 0.5443 - loss: 0.1585 - precision: 0.6280 - recall: 0.2462
accuracy: 0.6646 - auc: 0.6383 - f1_score: 0.5449 - loss: 0.1584 - precision: 0.6287 - recall: 0.2549
accuracy: 0.6649 - auc: 0.6401 - f1_score: 0.5442 - loss: 0.1582 - precision: 0.6275 - recall: 0.2547
```

Now, the focal loss still needed modifications in its parameters. Focal loss alpha form equation is:

$$FL = -\alpha * (1 - x)^{\gamma} * \log(x)$$

For positive class

$$FL = -(1 - \alpha) * (1 - x)^{\gamma} * \log(x)$$

For negative class

The best value for gamma was found to be 2, which is the default, so no change. The choice was to modify the value of alpha and a boolean parameter `apply_class_balancing`. For this dataset, alpha was computed using the function `compute_class_weight` whose equation for binary classes is:

$weight_{class} = \frac{n_{samples}}{2 * count_{class}}$, so it is inversely proportional to the count of the class, weighting a larger value to the minority class.

However, alpha value needs to be between 0 and 1 from the equation mentioned earlier; thus, class weights were normalized by the following formula:

$$weight_{class} = \frac{weight_{class}}{\sum_{class} weight_{class}}$$

Finally, alpha was set initially to be equal $weight_{pos_label}$ to penalize the misprediction of the minority class more than the majority class. Then, 2 models were trained on this loss function, but one when the boolean parameter = False, and one when True.

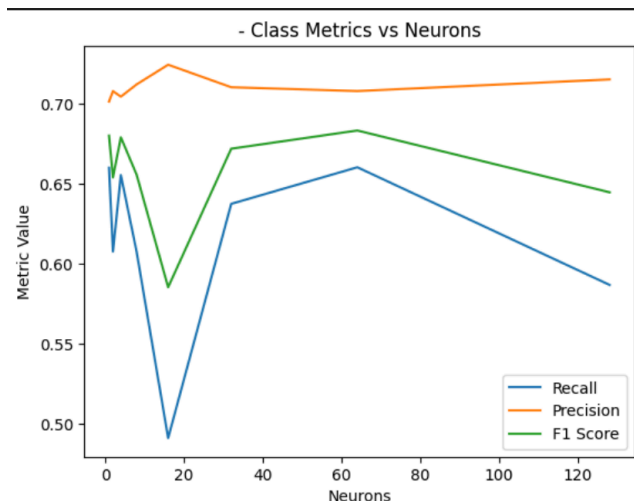
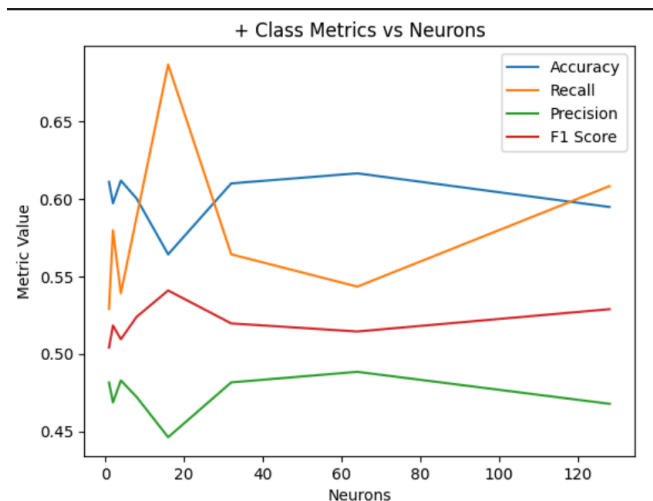
Results → This was a pivotal point in this project as the metrics started to be fairly acceptable when the binary parameter = True, and alpha was set by prior computations. Metrics were better than all metrics resulting from milestone 3 as recall and f1-score for the positive class are now around 0.44 and 0.47 respectively. They are still not efficient, so more tuning was done on the loss function by trying different values for alpha starting from 0.65(as it was 0.62) to 0.8 increasing by 0.05 each time. To decide the best out of them, the formula shown below was manifested. The metric is simply the weighted average of the F1-score for both classes.

$$Metric = weight_{pos_label} * avg(F1score_{pos_label}) + (1 - weight_{neg_label}) * avg(F1score_{neg_label})$$

Results → alpha = 0.65 was found to be the best one as the more it increases after this value, the more extremely biased the model gets towards predicting positive classes, making the positive class recall values near 80% and 90%, but the recall for the other class is terrible, near 20% for example.

NN Architecture

After settling the loss function and its parameters, the goal was to pick the best architecture for the neural network. When trying 1 hidden layer with all activation functions sigmoid, the following two plots show the variation of metrics in relation to perceptron number.



From the plots, there is a close call between having 16 or 32 neurons, but increasing more than this is unnecessary as no critical improvements are shown. Thus based on these results, continuation of tuning the hidden layer would be by trying also having 16 or 32 perceptrons, but the activation function is ReLU.

Results → Although the performance results of the four combinations are close to each other as shown in the picture below, what matters is the metrics relevant to the purpose of this project. For that reason, the optimal choice would be 16 sigmoid neurons because of the following analysis:

- It has the highest positive class recall = 0.686 with a difference from the others (0.62 the second highest)
- It has the highest positive class F1-score = 0.54
- Although it has the lowest positive class precision = 0.446; However, the highest recorded is 0.48, so it is not a critical difference

- Although it does not have a very good negative class recall value, but in real-life situations positive class recall is very important because a misprediction of a healthy human as a COVID carrier is not less critical than mispredicting a COVID carrier to be healthy, so it is a matter of balancing and knowing the most important metrics at the end since they are not the best overall

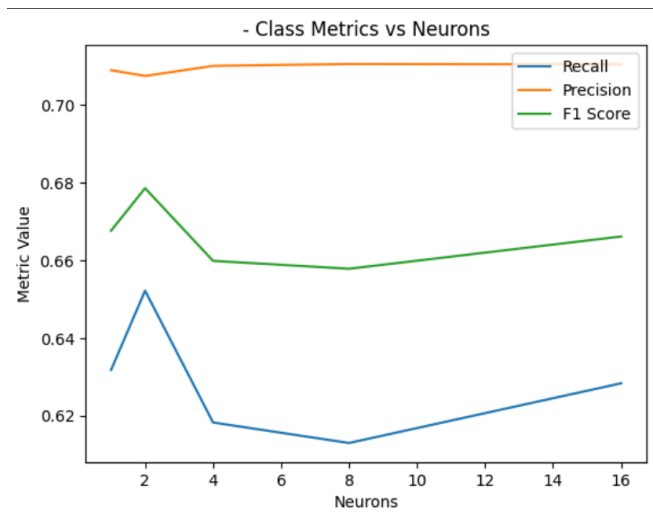
16 ReLU	Precision	Recall	F1-score	Accuracy
Class +	0.4688002316312695	0.6003128244303886	0.5218822744940957	0.591404525188947
Class -	0.712488024170114	0.5863636481570024	0.6372313616808452	
16 sigmoid	Precision	Recall	F1-score	Accuracy
Class +	0.4461968174827442	0.6867525662198917	0.5409359121130303	0.5642686503111366
Class -	0.7242442959343436	0.4911465053919038	0.5853425346573182	
32 ReLU	Precision	Recall	F1-score	Accuracy
Class +	0.4651751469367603	0.6295886455241136	0.5219612698715466	0.5764489902963545
Class -	0.7172867706093063	0.5447272984427338	0.5992461009811684	
32 sigmoid	Precision	Recall	F1-score	Accuracy
Class +	0.4015892122310076	0.5642933924200774	0.5196697986444921	0.6100517368810052
Class -	0.710177272441209	0.6373695010178292	0.6718055785306771	

Second Hidden Layer

As mentioned in the earlier report, a second hidden layer was experimented with fixing the first to be 16 sigmoid perceptrons. The second hidden layer was experimented with neurons = 1, 2, 4, 8, and 16, all being sigmoid.

Results → All neuron values did not yield better performance metrics as the positive class F1-score dropped and its recall also dropped, which are the two most important metrics. The plotting is attached below for reference.





Optimizer

All the prior models were trained using Adam as an optimizer, so SGD and Adagrad were experimented on the best fine-tuning induced yet.

Results → The Adagrad optimizer was extreme in its results, making the negative class recall = 0.16, and SGD was less extreme than Adagrad, but still did not yield better results than Adam, so Adam was found as the best optimizer among the three.

Regularization

L1 and L2 regularization were added after, but the results were worse, which is due to the fact that they are used to reduce overfitting, but this model is not overfitting at all as the validation metrics are similar to training, and they are actually not high to be considered in overfitting.

Smote

A final trial was to do oversampling using smote method, and normal binary cross entropy loss function was tried, and then focal loss.

Results → The results of binary cross entropy were not good, while the focal loss was not of an improvement; they were similar to some of what was conducted before.

FINAL MODEL HYPERPARAMETERS

The final model hyperparameters are:

- **1** hidden layer
- **16** neurons
- **Sigmoid** as the activation function for all perceptrons
- **Learning Rate** = $1e-3$
- **Optimizer** = Adam
- **Early Stopping:**
 - Monitored value = F1-score
 - Patience = 3 (how many worse epochs in F1-score the model will wait before stopping)
 - Restore the best weights in case of early stopping
- **Epochs** = 10
- **Batch Size** = 2048
- **Loss Function** = Focal Loss with $\alpha = 0.65$, and `apply_class_balancing = True`

APIS

The primary service offered by the website is COVID state prediction, which is initiated when a user navigates to the COVID form, completes it, and submits it. The API then processes the user input using the Keras libraries and dependencies to generate a prediction. This prediction is returned to the views component of the Django project as a boolean value indicating the likelihood of the prediction being true or false. Subsequently, a new webpage prompts the user to provide the ground truth value of their state for model retraining. If the user declines to provide real feedback, their data entry is not saved in the dataset.

During filling in the form by the user, there are many restrictions that guide the user to successfully fill in the form like: if the user chooses the sex to be male, thus s/he cannot choose to be pregnant. Moreover, there is a default value on each of the form options that forces the user to initially choose a value at each and every attribute.

Everytime the user goes to the homepage, a request occurs at the backend asking whether there were more than X number of users filled in the form or not. And that X number is predefined by the admin of the project according to many attributes. If yes, meaning the number of entries exceeded X, the backend of the project retrains the model using the dataset used and then deletes them, so that the new

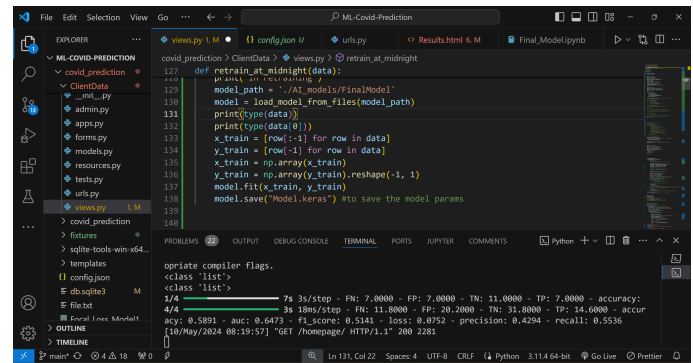
prediction is more precise on more and more data inputs.

PREPROCESSING

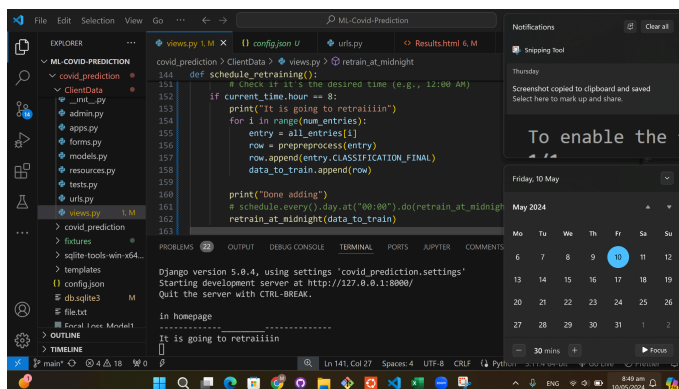
For the current dataset, since most features are binary already, and Django takes the input as boolean values already. The only numerical feature is age, and it does not need processing as well. Hence, the data are taken and created as a list of np arrays, and sent to the model for prediction.

RETRAINING

Leveraging the small size of the dataset nature, the whole dataset was uploaded into a local instance of Django locally at first, then the complete Django instance was deployed on PythonAnywhere with deploying all of its dependencies as well. If a patient gives feedback on the ground truth, then their medical record is saved and concatenated on the database of Django internally. Then, based on a specified time daily, which we set to be 12:00 am, the backend will stop freeze the website, and the model is going to retrain on the whole dataset and the server would not be working. However, for time purposes, the whole dataset was not uploaded, but instead almost a one 5-fold of it. After the model finishes retraining, the new weights will be saved overwriting the existing ones, and the website will go back to working normally. In other words, the website is working everyday any time except for one hour between 12:00 am to 1:00 am for retraining. Attached below is an example of an hour specified as 8 to show that the model can be retrained.



```
def retrain_at_midnight(data):
    # Load data from CSV
    model_path = './AI_Models/FinalModel'
    model = load_model_from_files(model_path)
    print(type(data))
    print(type(data[0]))
    x_train = [row[-2] for row in data]
    y_train = [row[-1] for row in data]
    x_train = np.array(x_train)
    y_train = np.array(y_train).reshape(-1, 1)
    model.fit(x_train, y_train)
    model.save(model_path) # to save the model params
```



```
def schedule_retraining():
    # Check if it's the desired time (e.g., 12:00 AM)
    if current_time.hour == 8:
        print("It is going to retrain!!!")
        for i in range(num_entries):
            entry = all_entries[i]
            row = preprocess(entry)
            row.append(entry.CLASSIFICATION_FINAL)
            data_to_train.append(row)
        print("Done adding")
        # Schedule every 1 day at 12:00 AM
        do(retrain_at_midnight, data_to_train)
```