ORML: Project 1

Rory Kolster, SNR: 2070539, ANR: 736326

November 2024

1 Neural Network

To alter the design and tuning of the hyperparameters of the neural network (NN), I ran training and validation, then from observing their accuracy and error loss I made adjustments. Initially, I focused on the NN architecture regarding nodes and layer design. I planned on fine-tuning hyperparameters like the optimization algorithm, learning rate, and activation functions later, once the architecture was desirable. As for the initialization, I decided to use the automatic initialization used by keras, as they already use a relatively smart one.

To obtain a starting point, I used the same architecture used for the MNIST dataset from the template adding some changes, with the following design: 2 hidden layers, $[512, 512]$ nodes, batch size of 32, no regularization, RMSprop optmizer, learning rate of 0.003, and only 10 epochs. Naturally, this yielded a poor accuracy of 52%. Hence, the following attempts changed the number of nodes and layers and increased epochs to 100. I used 3 hidden layers with $[200, 50, 10]$ nodes. The resulting accuracy for training was 96%, while validation was 78%. This is clear overfitting, and hence the following changes were to address that.

To address the overfitting, I considered Srivastava et al., 2014 and implemented dropout in the input layer, using the ready-made functions in keras, and set the rate to 0.5. This resulted in a training accuracy of 72% and a validation of 85%, which was surprising. However, the issue of overfitting seemed to be dealt with. The following attempts adjusted the number of nodes and layers again, and adding dropout to the hidden nodes as well. For the nodes, I went back and forth between 4 and 5 layers with $[200, 100, 50, 10]$ and $[400, 200, 100, 50, 10]$ nodes, respectively. This was to use the "halving" number of nodes per layer strategy, to mimic dropout. These implementations seemed not to change the performance much, if anything they were slightly worse.

Although, then I tried to introduce dropout for the hidden layers manually through the keras functions. At this point, I realised that it was recommended to have 0.8 dropout rate in the input layer, with 0.5 rate in the hidden layers. However, implementing this resulted in very poor performance, restricting both training and validation accuracy to 50%. This makes sense as it is now underfitting.

To this effect, I returned to the best performing hyperparamters thus far, which was 4 layers, $[200, 100, 50, 10]$, learning rate of 0.001, and a 0.2 dropout rate in the input layer. I began adjusting the batch size as I still noticed "jumpy" behaviour in the accuracy and error graphs. I also noticed that learning rate and batch size were inextricably linked and so began choosing the best combination. Despite this, I still continued, for now, with the same learning rate of 0.001 and a batch size of 32. The following figure 1 shows the error loss for the various batch sizes.

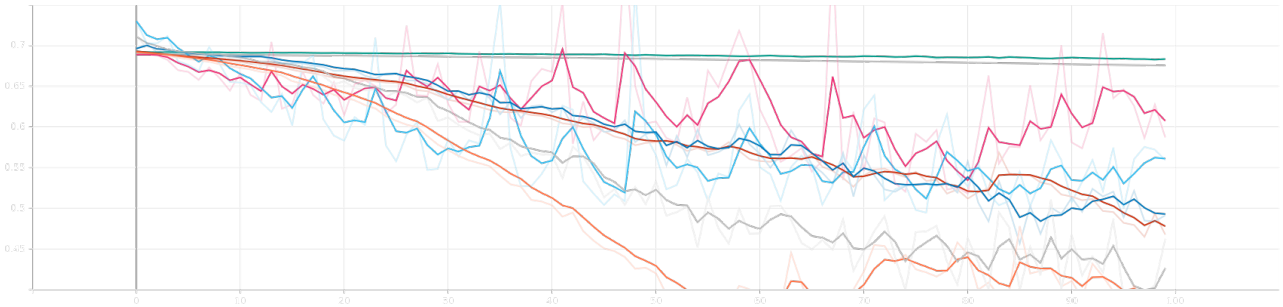

Figure 1: Loss per Epoch, comparing Batch sizes

Now I began a grid search to find the best dropout rate for the input layer. I tested $[0.05, 0.1, 0.15, 0.2, 0.25]$. The best performing rate was 0.05 which is to be expected as it potentially allows for overfitting, but it was still not too much overfitting. The following figure 2 illustrates the error loss for the gridsearch of the dropout rate.

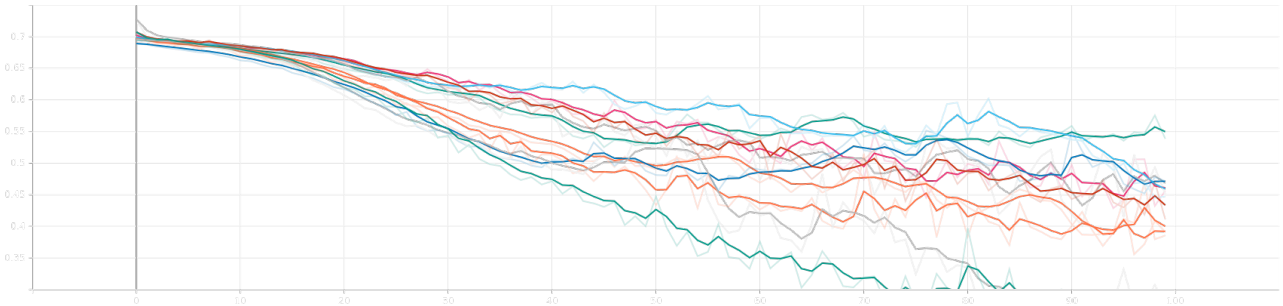

Figure 2: Loss per Epoch, comparing dropout rates

Then I attempted to determine the best optimizers for my NN. Since the optimizers are sensitive to various learning rates, I tested each for some ranges. It seemed that RMSprop performed the best, with Adam producing decent results as well. This might be because I had optimized the NN up to this point only working with RMSprop. Subsequently, I then looked at adjusting the momentum and rho parameters for RMSprop, but it seemed the standard inputs were optimal already. Returning to the best acheived design thus far, I noticed that the training was again out-performing the validation. In response, I increased the dropout rate to 0.1, which closed the gap, especially in the error loss. The below figure 3 shows the accuracy over epochs for the optimizer RMSprop for different learning rates.

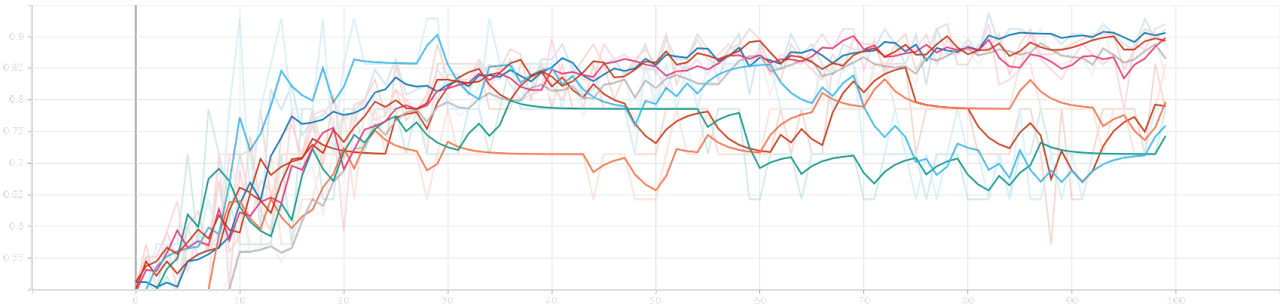

Figure 3: Accuracy per Epoch, for RMSprop and various learning rates

Now I looked at altering the activation functions. According to the literature: "In modern neural networks, the default recommendation [hidden nodes] is to use ... ReLU" page 174 Ian Goodfellow, 2016. It is also advised depending on the type of problem (regression, binary classification, multi classification) to go for sigmoid or softmax. Again I tried each combination, implementing Leaky ReLU and Tanh for the activation function of the hidden nodes and sigmoid and softmax for the output nodes. As expected, softmax performed poorly, so I opted for sigmoid. But surprisingly Leaky ReLU underperformed compared to sigmoid.

Finally, using the testing of the above features, This left me with the following NN design:

- 4 hidden layers with [200,100,50,10]
- batch size: 32
- dropout regularization in input, with rate 0.05 with hidden dropout by manual design
- RMSprop optimizer
- learning rate: 0.001
- Hidden activation function: Sigmoid
- Output activation function: Sigmoid
- automatic initialization

The following figure 4 shows the performance of the training and validation for accuracy and error loss.

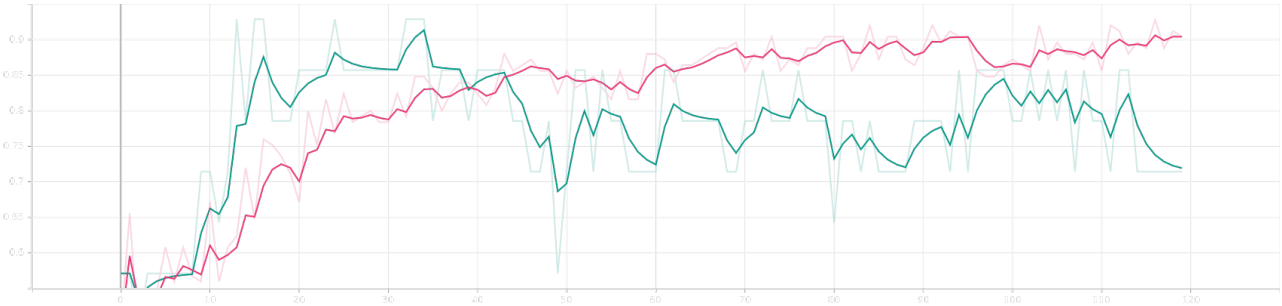

Figure 4: Accuracy per Epoch, for the final NN

What posed quite a significant challenge, was the fact that the keras function that splits the data into training and testing was random. This meant that sometimes a given split of the data would yield a high accuracy based on a certain change, despite the change being an actual beneficial one. Hence, I would often return to the best acheived design so far, and trying to replicate this was often not possible. A possible solution to this is to of course set a random state seed in python, however, this then means you are potentially overfitting to that specific split of the dataset.

2 Local Search based Algorithm

In this question, I developed a local search inspired heuristic algorithm. I had threshold values τ_j and a binary variable s_j which was 1 if AND clause j was an upper bound and 0 if it was a lower bound. I split the data set of 145 datapoints into a training set of 70% $\approx$ 122, a validation set of 20% $\approx$ 35, and a test set of 10% $\approx$ 17. I use two objective values to minimize, in iterations. The first is the number of misclassifications, i.e.

$$min \sum_{n:y_n=0} \hat{y} + \sum_{n:y_n=1} 1 - \hat{y}, \tag{1}$$

and the second is balanced accuracy, where I use the function from *sklearn*. The initialization of the algorithm is a random split of s_j, i.e. half of the features are upper bounds and half are lower bounds, selected at random. Then the τ_j are set as follows. For instance, if $s_j = 1$, then the threshold for that j is an upper bound and then tau_j is set to the maximum value of X across the dataset (in this case the training set), for feature j. If $s_j = 0$, then τ_j is set to the minimum. This means that the algorithm at the end of phase 1 predicts all patients correctly, with some healthy predicted incorrectly as patients. The local search method had 2 phases:

1. Phase 1:

    - The algorithm searchs through neighbourhoods of changing s_j. One s_j was changed per solution in a neighbourhood, and the tau_j were changed according to the value of s_j, in the same manner as explained above.
    - In this phase, I minimize with respect to the misclassifications objective. Usually this phase starts with an objective value of between 3 and 6, and then iterations lower it towards between 0 and 3. Sometimes it yields zero, and does so within 5 iterations or so, but other times it remains one number for many iterations. Thus after, at least 10 iterations if the same objective value is repeated 5 times, I proceed to phase 2. For a visual, refer to Figure 5

2. Phase 2:

    - This phase of the algorithm searches through neighbourhoods where it changes the threshold values. If τ_j is an upper bound, then it reduces it by a prescribed amount (in the code I use *tau steps*), and increases it if it is a lower bound. The attempt with this part is to prevent the overfitting to the training data, and increasing the balanced accuracy by predicting fewer healthy people as patients.
    - The idea is to take *tau len* number of solutions after iterating through *phase2 depth* new thresholds, and then test these solutions on the validation set, and then choose the "best" one from this. I choose the best ones based on the balanced accuracy objective instead of the misclassifications objective.

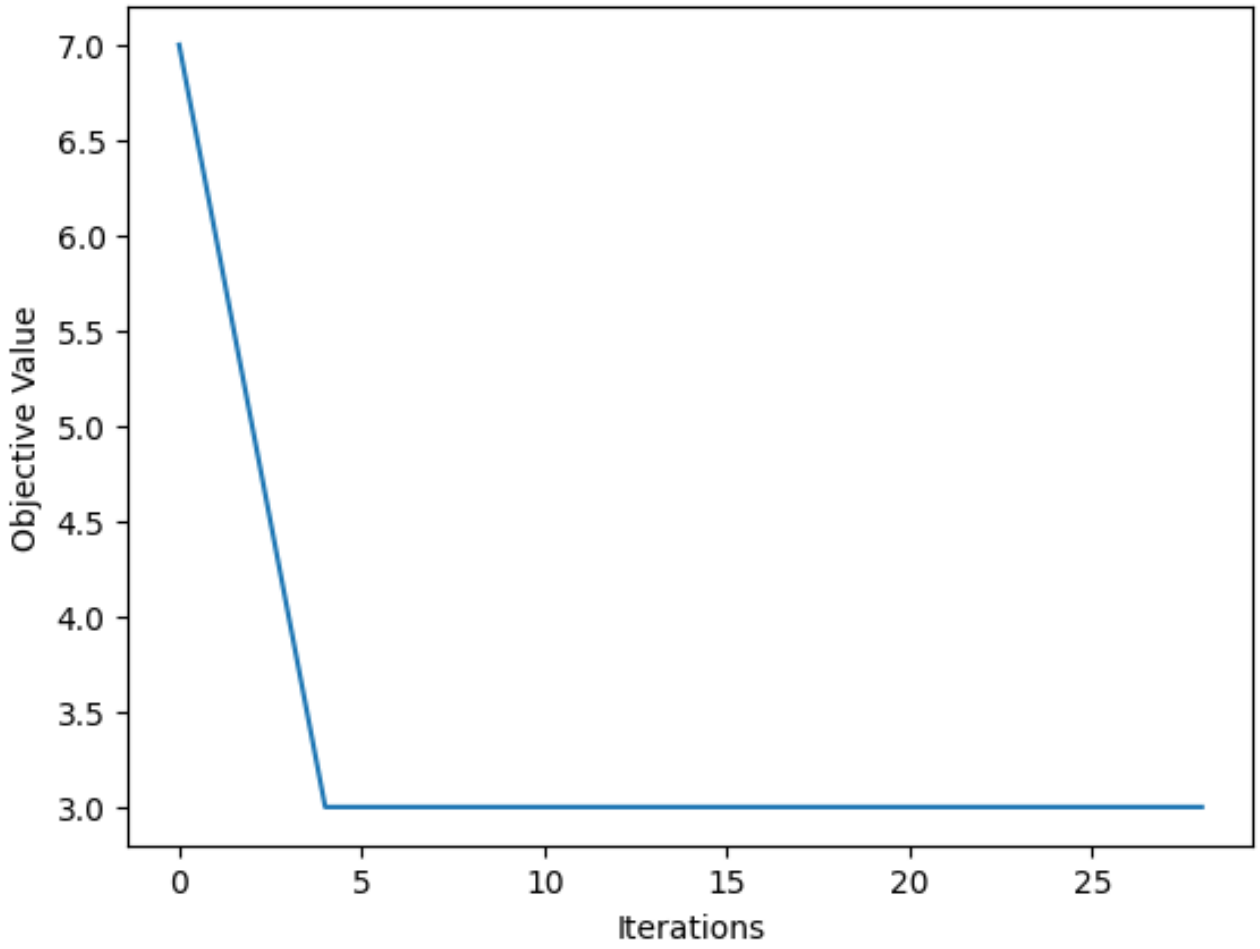

Figure 5: objective value, per iteration in phase 1

If at the end, phase 2 did not improve upon the solutions from phase 1, the algorithm returns a solution from phase 1. There are a few hyperparameters to be considered, when running this algorithm. *tau size* determines the step size of the increments by which the threshold values decrease or increase in phase 2. Increasing this just means you take smaller steps so are less likely to change the predictions between iterations. *tau len* refers to the number of solutions that you take from phase 1 through to phase 2. Naturally, increasing this increases your chances of obtaining a better solution in the end, but it also increases your neighbourhood size in phase 2, causing your running time to increase. *phase2 depth* is how many iterations you allow phase 2 to go through, meaning how many times you decrease/increase the threshold values.
Neglecting running time as a factor, *tau len* and *phase2 depth* would ideally be larger, whereas *tau size* should be fine-tuned to the data, which can be potentially difficult.

A potential result of the algorithm could be as the following figure 6.

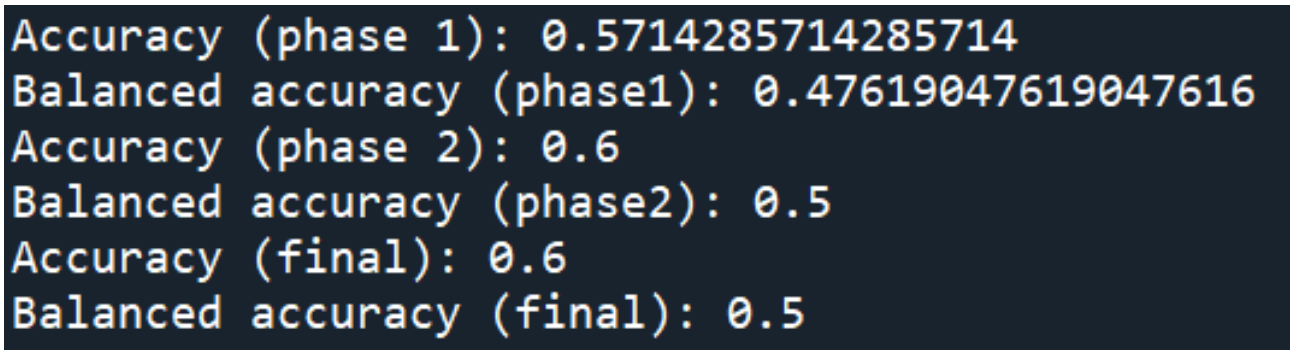

Figure 6: Algorithm Output

3 Method Comparison

After training the neural network and testing it on the test set, the accuracy was (on a single run) 72%. Whereas, for the heuristic algorithm it was 50% (60% for misclassifications objective accuracy). So at first glance, it would seem that the neural network is superior.
The advantages of the neural network, is that it has the benefit of generality. It can be more easily applied to different scenarios, and it can perform potentially well despite non-representative data, assuming the NN does not overfit. It has been shown extensively in the literature that neural networks can be highly accurate at classification problems, for example Alex Krizhevsky, 2017 (AlexNet). However, the fact of overfitting is difficult to overcome as it is so easy to design to overfit. Especially on small datasets such as this, this is a problem. On the other hand there are tools to address this, like dropout. In general, neural networks require a lot of data to train and be effective, so applying neural networks to this dataset will not provide the outstanding results anyway.

For the algorithm, due to the strict and challenging nature of the problem definition of using the AND clauses, the effectiveness of the classifier produced is perhaps capped. Even if an optimal solution is found with an MILP, the "optimal" solution is likely to be only optimal for the training set. Furthermore, if the data is not representative, this also poses problems just like the neural network.

When optimizing the NN, what I found challenging was that when I made a change, and then observed the outcome of that change, it rarely had the expected or desired effect. This made making implementations that improved the network difficult.

For the heuristic algorithm, to improve further I would redesign the neighbourhoods into something that had more effect on the classifactions. I would also implement a faster and smarter method for calculating the objective value for each solution in the neighbourhood. I did it in a naive manner, which affected the running time severly. Although, in general, I recommend not implementing this local search method, as the neighbourhoods, at least as I designed them, were not effective. Instead, genetic algorithm or another would be more fruitful.

Therefore, in general, the neural network outperformed the heuristic algorithm. Also the advantages with the disadvantages of the NN outweigh the advantages and disadvantages of the algorithm. Also, if better designed I expect the neural network to be far more effective at classification than the relationship formulation. The performance of both methods can also be somewhat attributed to the size of the dataset, however it is likely that this is the largest problem regarding operations research in this field of healthcare and medication.

4 Appendix

References

Alex Krizhevsky, G. E. H., Ilya Sutskever. (2017). Imagenet classification with deep convolutional neural networks. *60*(6), 84–90.

Ian Goodfellow, A. C., Yoshua Bengio. (2016). *Deep learning*. The MIT Press.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, *15*(56), 1929–1958. http://jmlr.org/papers/v15/srivastava14a.html