CO395 - Introduction to Machine Learning
Neural Network

Amin-Nejad Ali, Gan Ding Han, Liu Lingjun, Yoon Goonhu
{aa5118, dhg18, ll5018, gy18}@imperial.ac.uk

Course: CO395, Imperial College London

8th March, 2019

1 Introduction

Neural networks, loosely based on the functionality and structure of the human brain, are powerful models able to capture and represent complex input and output relations. The core building block of a neural network is the aptly named neuron, groups of which are usually arranged into interconnected layers. They are most-commonly used for clustering and classification, but can be also used for regression tasks. During the learning phase, input data is passed through the whole network whilst an appropriate loss function at the output layer is calculated. This loss is then propagated backwards through the network to adjust the individual weights and biases of the neurons, by virtue of the chain rule of calculus, with the aim of minimising this loss and ultimately finding the best model.

1.1 Problem Overview

The first task we were given was to create our own neural network library from the skeleton files and then test it on the infamous `iris` dataset. Subsequently, for the second task, we were expected to train, test and evaluate a deep neural network on the forward model of the ABB IRB 120 robot i.e. predict the Cartesian position of the robot arm given the angular position of the first three motors. The final task instead required us to predict in which one of four zones the robot arm will be given the same data regarding the angular position of the first three motors.

2 Learning the Forward Model of a Robot

2.1 Implementation

We decided to use the mini-library we had developed in the first task to solve this regression task. However, given the plethora of hyperparameters and the lack of defaults that may come with some other well known deep learning python packages, we were initially unsure of where to begin. We ultimately decided to use our machine learning knowledge and reasoning skills to come up with sensible values as a starting point from which we can iterate upon if necessary.

Architecturally speaking, given the relative simplicity of the task, the abundance of data ($\sim$ 16k observations) and the small number of input and output parameters (3 each), we reasoned that the network need not necessarily be particularly deep or wide to perform highly. On this basis, we opted for a structure composed of **3 hidden layers of 8 neurons each**. For simplicity, we opted to have a constant number of neurons per hidden layer within a densely connected network. The initial activation function of choice was **ReLU** to avoid the *vanishing gradient problem* whilst **mean squared error** was chosen as the most sensible loss function given the nature of the task. The batch size chosen was **64** (a power of 2 for efficiency purposes) with an awareness of the trade-off between speed of convergence and the likelihood of convergence.

Lastly, for further simplicity, a constant learning rate of $\mathbf{1e^{-3}}$ was chosen instead of using a decaying learning rate or more sophisticated implementations. The number of epochs was left essentially unlimited with a view to stopping the training at the point of convergence. This design managed to achieve a normalised MSE loss of ~ 0.01 within ~ 1000 epochs.

2.2 Evaluating the Architecture

Given this was a regression task, there are a slightly more limited number of useful metrics to look at to analyse the performance of our model. The primary metric which is commonly used in this situation is *mean squared error* (MSE).

$$MSE = \frac{1}{n}\sum_{i=0}^{n}(y_i - \hat{y}_i)^2 \tag{1}$$

This metric is very useful in determining the absolute difference between the model prediction and the true values. However, there is no indication given for the directionality or skew of the error or much of a good fit the model is. Furthermore, MSE also disproportionately penalises larger errors compared to smaller ones. For this reason, we decided to also compute two other metrics, the *mean absolute error* (MAE) and the *coefficient of determination* (R^2).

$$MAE = \frac{1}{n}\sum_{i=0}^{n}|y_i - \hat{y}_i| \tag{2}$$

$$R^2 = 1 - \frac{\sum_{i=0}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=0}^{n}(y_i - \bar{y})^2} = 1 - \frac{MSE}{\sigma^2} \tag{3}$$

The MAE does not disproportionately penalise larger errors making it more robust to outliers whilst the R^2 provides a measure between 0 and 1 of the goodness of fit of the model prediction compared to the observed values ensuring that the model prediction also has a low dispersion. Thus, we decided to calculate all 3 metrics to analyse the performance of our model.

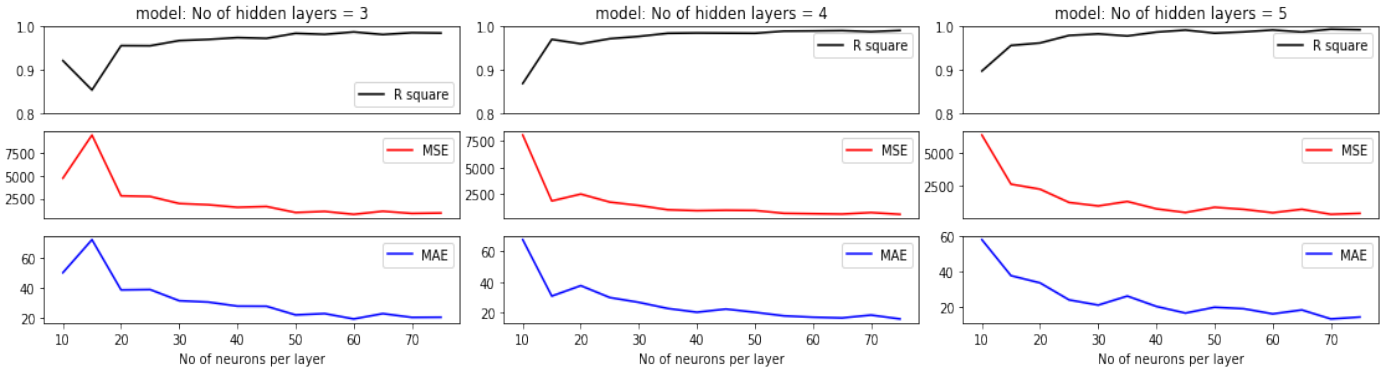

Figure 1: Un-normalised performance of models with different hidden layers(3, 4 and 5 respectively)

2.3 Fine Tuning the Architecture

Tuning the hyperparameters to their optimal values was also a difficult task. With so many hyperparameters and so many different permutations, it would be almost impossible to try and train a model exhaustively on each set of permutations in order to ascertain the optimal values for each hyperparameter. This approach would take a very large amount of time and computing power. On the other hand, changing too few hyperparameters would also be far from ideal due to the high number of interdependencies between the hyperparameters forming a complex web of dynamically changing

relationships. If one doesn't loop through a sufficient number of permutations, there is no indication as to whether or not one has found the optimal values. Therefore, our task was to try and find this middle ground where we have looked at enough permutations to feel reasonably comfortable with our optimal hyperparameters without performing an exhaustive search.

For this reason, we necessarily had to make some assumptions and simplifications. Firstly we decided to keep most of the variables constant while we fine tuned the number of layers and neurons via a mini grid search. We opted for a sensible number of hidden layers to experiment with - just **3-5** given the relatively simple nature of the task. The number of neurons per layer was kept constant between the layers but ranged from 10 to 50 by increments of 5 neurons. As illustrated in Figure 1, the performance of the model kept improving up to the maximum of 50 neurons per layer across all 3 metrics that we were computing on the test dataset. This result was replicated with both 4 and 5 hidden layers as well. However, following the principle of Occam's Razor, we decided to save the model with 3 layers to avoid overfitting. There is no need to have more layers if the loss is not decreasing any further and these extra neurons will in fact start to encode more complex, yet irrelevant, properties specific to just the training dataset.

After fine tuning the hyper-parameters, the MSE, MAE and R^2 reach 855.87, 21 and 0.98 respectively for the final architecture shown in Table 1. It is worth mentioning that these final metrics are calculated from the original(un-normalised) values. However, the MSE loss calculated during training is calculated from the pre-processed(normalized) dataset as depicted in Figure 2. The figure shows the MSE Loss change plotted against the number of epochs, showing that the model reaches its convergence point at approximately epoch 400. Given these strong results, we felt we did not need to experiment further with the learning rate, batch size or activation functions.

Hyperparameter	Value
No. of hidden layers	3
No. of neurons per hidden layer	50
Activation functions for hidden layers	ReLU
Activation function for output layer	Linear
Loss function	MSE
Batch size	64
Learning rate	1e-3
No. of epochs	1000

Table 1: Final architecture for learning the "Forward model of a robot"

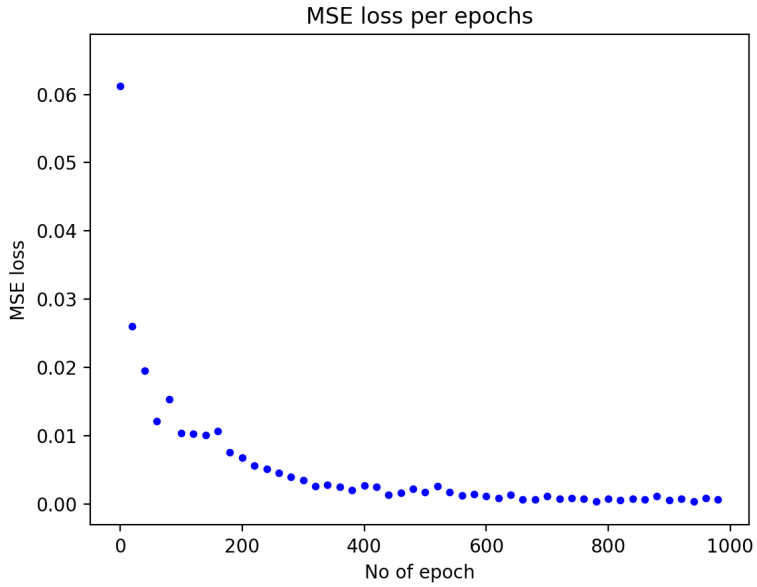

Figure 2: Normalised MSE Loss vs Number of epochs of fine-tuned model

3 Learning a "Region of Interest" Detector

3.1 Implementation

Learning a "Region of Interest" (ROI) was a classification problem, which required a slightly different approach compared to learning the Forward Model of the robot in the previous section. For one, the output of the neural network was no longer a set of regression values; for each set of inputs, the neural network would now output a series of 4 values corresponding to the 4 ROIs (henceforth referred to as "labels"). The label with the highest value would then be chosen as the overall output of the neural network given that set of inputs. With that in mind, slight modifications were made to the functionality of the previous section's neural network to allow it to predict and classify inputs into labels.

An initial architecture similar to that in the previous section was used in order to have a preliminary idea on the network's performance. The following were the hyperparameters used in the initial architecture:

1. No. of hidden layers = 3
2. No. of neurons per hidden layer = 20 (kept constant for all hidden layers for now)
3. Activation functions for hidden layers = ReLU
4. Activation function for output layer = Sigmoid
5. Loss function = MSE
6. Batch size = 64
7. Learning rate = 1e-3
8. No. of epochs = [start = 1000, end = 6000, increment = 1000]

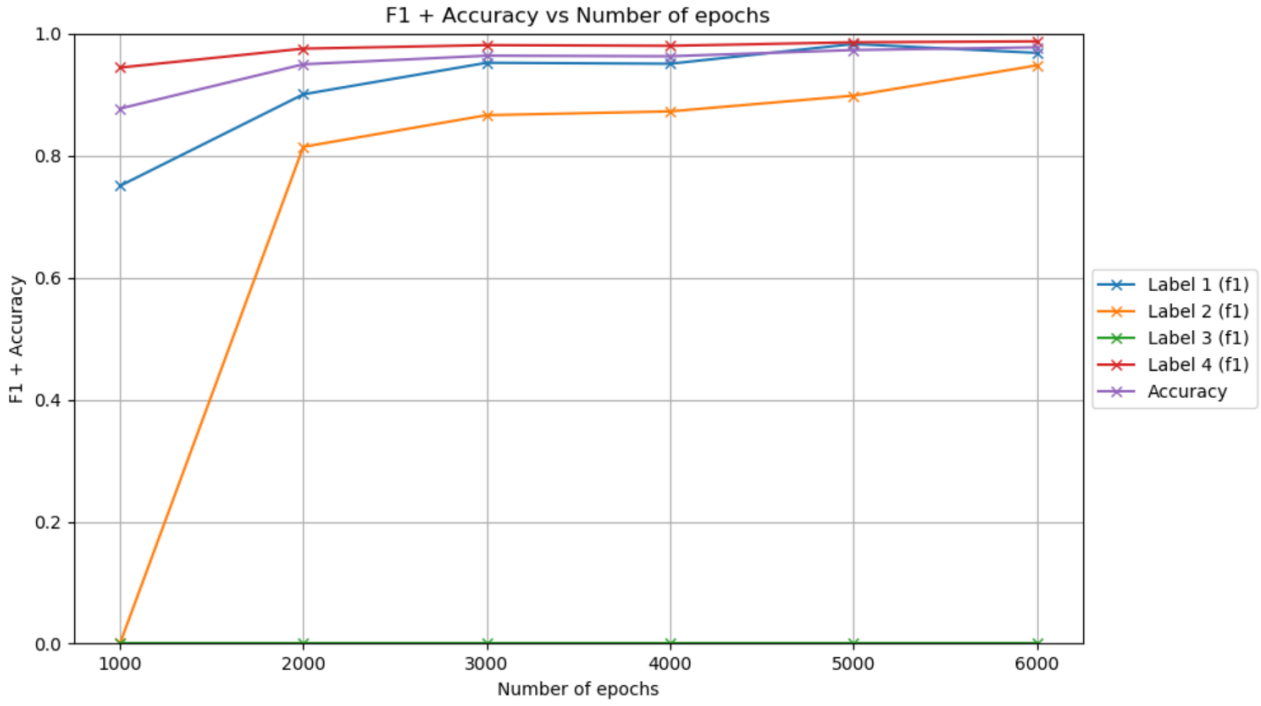

Figure 3: F1 Scores + Accuracy vs Number of epochs with Initial Architecture on Original Dataset.

There were strong justifications for setting the initial hyperparameters to those listed above. We believed that a relatively small number of hidden layers and neurons per hidden layer was appropriate

given that this problem did not appear complex, having only 3 input variables. Furthermore, ReLU was chosen for the hidden layers in order to avoid the vanishing/exploding gradients problem, whilst a sigmoid function was appropriate for the output layer so that we could conceptualise the outputs of the network as a series of probabilities for each label, with each "probability" between [0, 1]. We then kept the loss function, batch size and learning rate to be the same as that in Section 2 because they were still appropriate. However, we first vary the number of epochs to ensure that the loss function of the neural network does indeed converge. Thus, after setting the hyperparameters we then plotted the results as shown in Figure 3.

Figure 3 shows a plot of the F1 scores of each label, as well as the accuracy of the entire network, plotted against a variable number of epochs. The rationale for using these metrics is described in Section 3.2. One major takeaway from this figure is that the F1 score for label 3 is 0 throughout all epochs. This means that the network failed to make even a single prediction for label 3, which severely limits its usefulness. Indeed, analysis of the original dataset showed that only <1% of data belonged to label 3, whereas 80% of data belonged to label 4, thus explaining the skewed results.

In order to rectify this, we augmented the dataset by over-sampling to equally balance the proportions of data for each and every label. Specifically, we took reference from the label with the largest number of observations, which ensured that we did not discard any data at all. The original training data contained approximately 10,000 data points for label 4, which was by far the majority. Therefore, in order to over-sample each and every label in equal proportions, we augmented labels 1, 2 and 3 such that they each had 10,000 data points as well. We also added an element of Gaussian noise to all the duplicated data in order to allow the network to be more robust in its predictions. The Gaussian noise was added as such:

$$X^{'} = X + N(\mu, \sigma) \tag{4}$$

Our philosophy for picking appropriate μ and σ was that the noise variable should not result in a new data point which is an outlier compared to the original dataset. Therefore, to err on the side of caution, we set μ to be 0 and σ to be 10 raised to a magnitude. This magnitude was determined by selecting from the original dataset the smallest magnitude, minus 1. This would, ideally, provide the network more training data to make predictions of the under-represented labels.

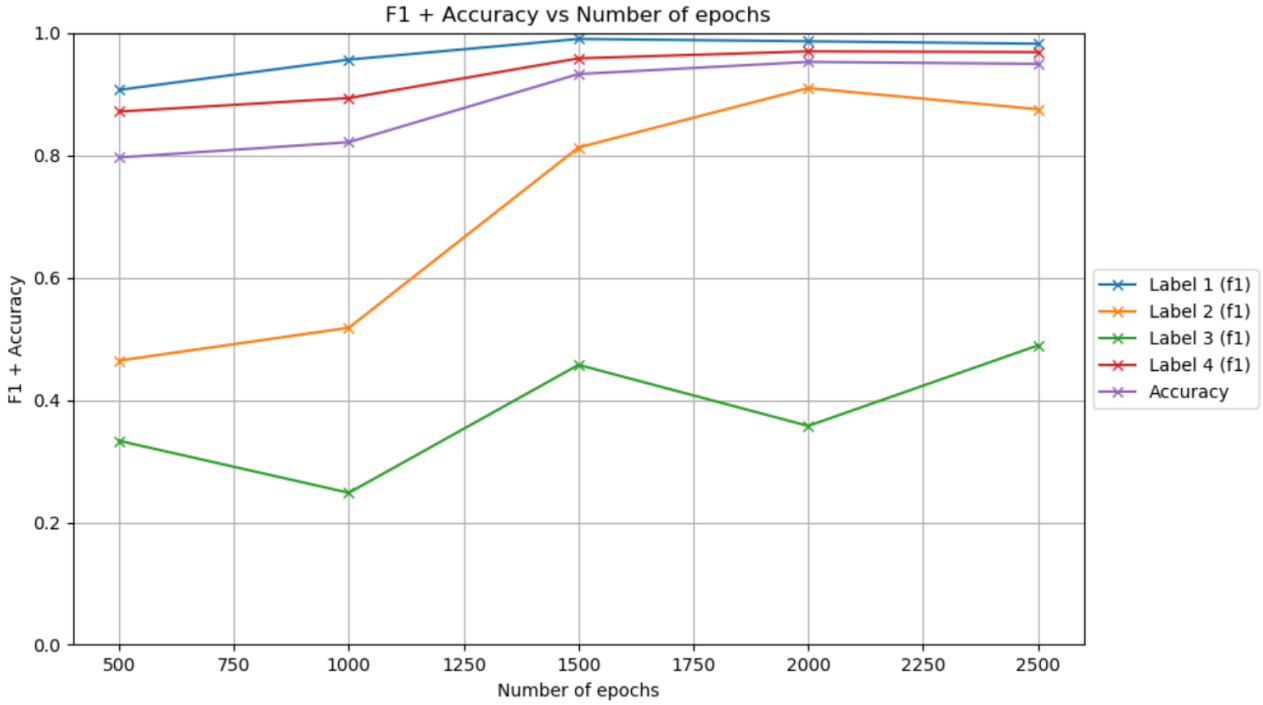

Figure 4: F1 Scores + Accuracy vs Number of epochs with Initial Architecture on Augmented Dataset.

Figure 4 shows the results of the network's performance using the augmented dataset. While the

F1 scores for labels 1, 2 and 4 and the accuracy show a slightly lower performance compared to Figure 3, there was a clear and obvious improvement in the F1 score for label 3. Therefore, as a summary of the implementation, moving forward we would use the augmented dataset to train the network while using 20% of the original dataset to test the network.

3.2 Evaluating the Architecture

After having seen some of the metrics used to evaluate the performance of the network in Section 3.1, we shall formalise the metrics to be used to fine-tune the architecture and state the rationale for using them. Since this is a multiclass classification problem, we want to investigate in detail the performance of the network in predicting each and every possible label. The best proxy for doing this is the F1 score, which encapsulates both the recall and precision scores for each label. We could thus use the F1 score to confidently state the performance of the network in predicting each label. In addition, we used accuracy as a general-purpose metric to evaluate the entire performance of the network. It is appropriate to use accuracy as a metric because we augmented the dataset to contain equal proportions of each label, whereas with the original dataset accuracy would be a meaningless metric due to the severely imbalanced nature of the dataset. In summary, we used the F1 scores for each label, as well as the accuracy to evaluate and fine-tune the network.

3.3 Fine Tuning the Architecture

As described in the previous subsections, the first step in fine-tuning the network was to pre-process the data by augmenting it to have equal proportions of each label. Thereafter, based on the initial architecture described in Section 3.1, we could already fix a few hyperparameters for the network. Based on Figure 4, we can derive that a reasonable number of epochs to use when fine-tuning the other hyperparameters such as number of hidden layers and neurons per hidden layer is approximately 2500. This is because at 2500 epochs, the F1 scores and accuracy are at a significant level, which means the loss during the training of the network was converging. Therefore, we can set some of the initial hyperparameters to be as such:

1. No. of hidden layers = [start = 3, end = 5, increment = 1]
2. No. of neurons per hidden layer = [start = 10, end = 50, increment = 5]
3. Activation functions for hidden layers = ReLU
4. Activation function for output layer = Sigmoid
5. Loss function = MSE
6. Batch size = 64
7. Learning rate = 1e-3
8. No. of epochs = 2500

Using these initial hyperparameters, we then had to tune the number of hidden layers and the number of neurons per hidden layer. We decided to opt for the same methodology as in Section 2.3. The range of hidden layers used was between 3 and 5 inclusive, which we felt was sufficient because the complexity of the problem did not call for an exceedingly deep network. In a similar vein, the range of the number of neurons per hidden layer was between 10 and 50 inclusive, which was kept constant for all hidden layers. The results of the tuning are shown in the figures below.

Comparing the figures in Figure 5, it is clear that the accuracy as well as the F1 scores for labels 1, 2 and 4 were consistently high ($\geq$ 80%) throughout all variations in the number of neurons per hidden layer. However, there were key differences in the performance of the F1 metric for label 3 across the different numbers of hidden layers used. Comparing the 3 figures, we can tell that having 4 or 5 hidden layers resulted in higher F1 scores for label 3 in general compared to having only 3

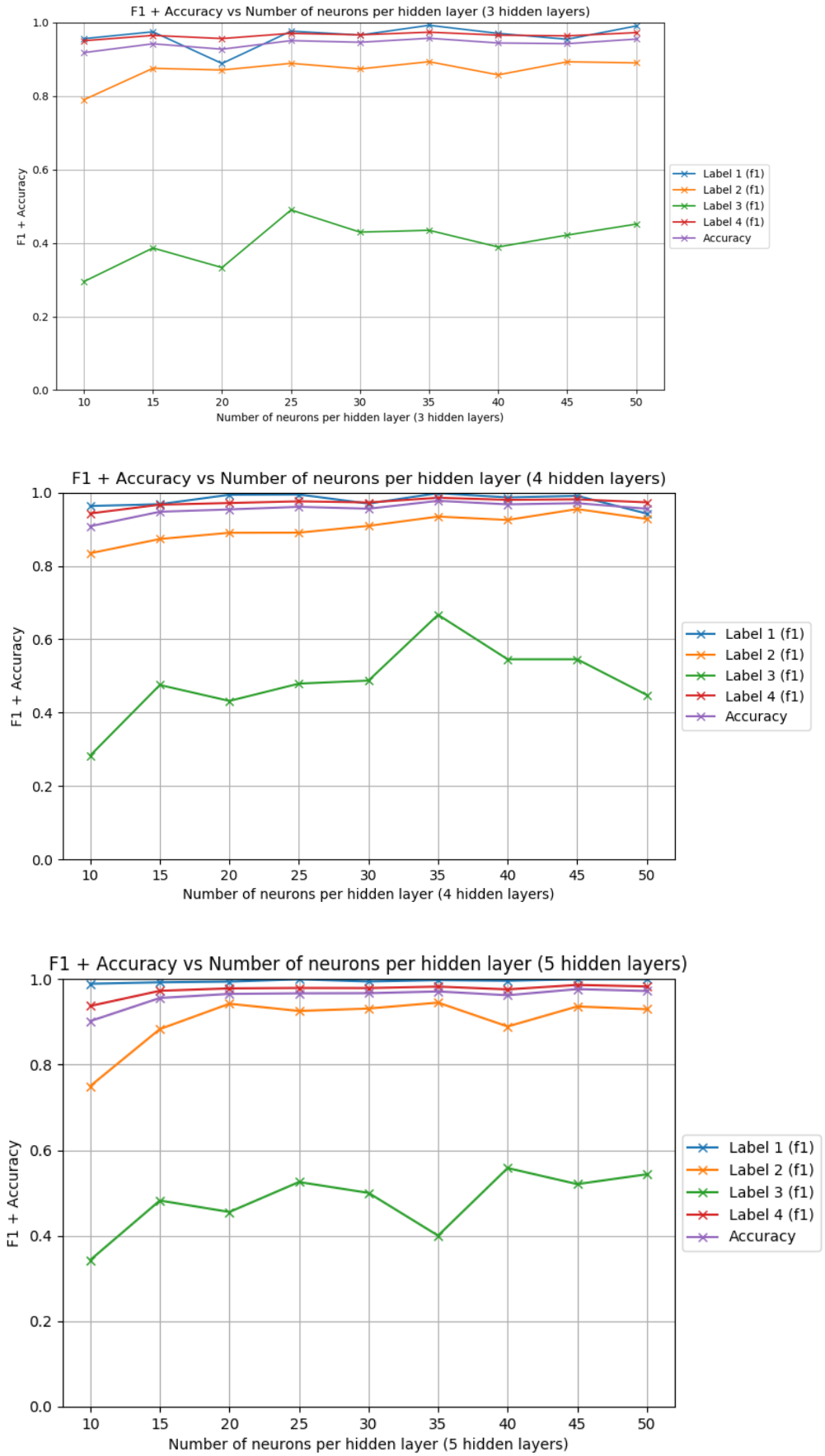

Figure 5: F1 Scores + Accuracy vs Number of Neurons per Hidden Layer (3, 4 and 5 Hidden Layers respectively) on Augmented Dataset.

hidden layers. Then, comparing 4 and 5 layers, we can conclude that although they both displayed similar performances in the F1 score for label 3, having 35 neurons per hidden layer with 4 hidden layers produced by far the best performance for F1 in label 3. Therefore, we can conclude that the final architecture for learning the "region of interest" should be as that depicted in Table 2.

Hyper-parameter	Value
No. of hidden layers	4
No. of neurons per hidden layer	35
Activation functions for hidden layers	Relu
Activation function for output layer	Sigmoid
Loss function	MSE
Batch size	64
Learning rate	1e-3
No. of epochs	1500

Table 2: final architecture for learning the "region of interest"

Finally, we also experimented with cross-entropy as the loss function which is traditionally more commonly used in classification tasks. However, empirically, the classification result tended to be less robust than using MSE as the loss function. As demonstrated in Figure 6, this result surprised us and led us to keep MSE as the loss function in the final architecture.

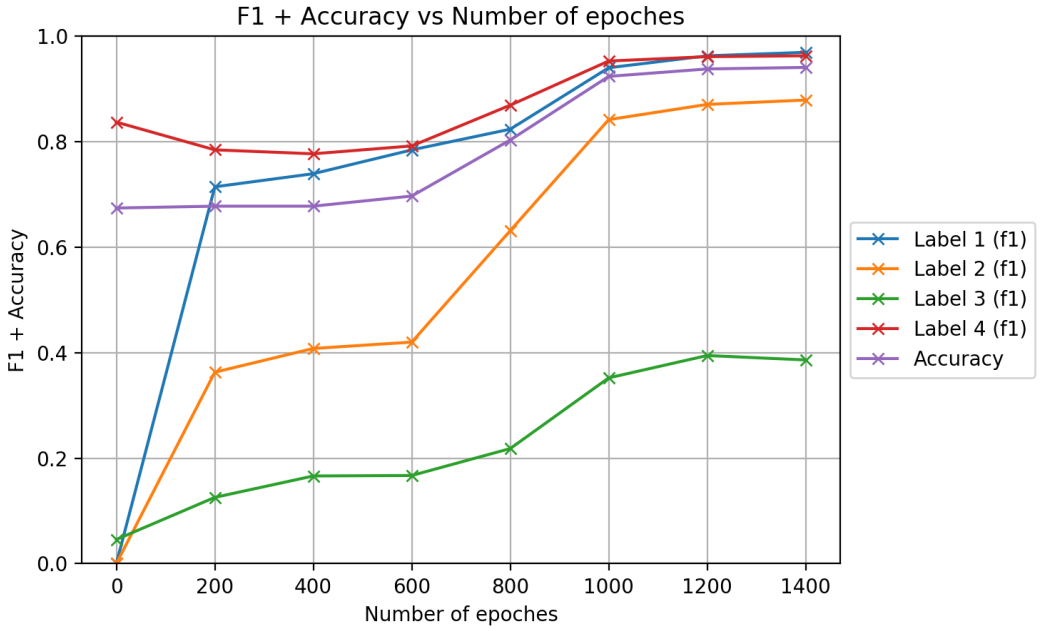

Figure 6: F1 Scores + Accuracy vs Number of epochs with Cross-Entropy loss function under the architecture as Table 2.

4 Conclusion

In summary, we have demonstrated how to build a neural network and employ it to tackle both the regression and classification problems. We have also demonstrated that by re-sampling (either over-sampling or down-sampling) the imbalanced dataset, the model can achieve better generalization performance.

In part 2, we show that by constructing a 3-layer neural network, with ReLU activation functions in all hidden layers, the performance of our regression model is great in terms of MSE, MAE and R^2

metrics. It is worth mentioning that both the input and target dataset are normalized during the pre-processing stage, for the sake of numerical stability in further training.

In part 3, the imbalanced distribution of the collected dataset makes it challenging to classify zone 3, to which less than 1% of the whole dataset belong. Instead, the model tends to over-predict zone 4, the category which covers more than 80% of the whole dataset. In order to tackle this problem, the over-sample and down-sample strategy is utilized, by which the generalization performance of our model is significantly improved. Lastly, we empirically show that this performance is more robust with an MSE loss as opposed to the more commonly employed cross entropy loss.