UNIVERSIDAD POLITÉCNICA DE MADRID

POLITÉCNICA

Deep Learning

Building a Deep Neural Network

MIGUEL PÉREZ - miguel.perez.mateo@alumnos.upm.es
DAVID BURRELL - d.burrell@alumnos.upm.es
IGNACIO MARTÍNEZ - ignacio.martinez.capella@alumnos.upm.es

Master's Programme in ICT Innovation: Data Science
&
Master of Science in Health & Medical Data Analytics (EIT Health)

Wednesday, 15 April 2020

Contents

1 Introduction

For this assignment our objective was to develop a neural network able to classify the proximity of a dwelling to the ocean. To perform this we were provided a dataset containing information about 20.427 dwellings within California, United States of America. The network can describe this distance using one of 4 classes: Near bay, Near ocean, Inland and <1H Ocean. The ground truth classification for the dwellings was provided in a dataset called OceanProximityOneHotEncodedClasses.csv, the classes being One-hot encoded. The encoding was represented with four binary value columns, one for each category, where each sample could only have a TRUE value in one column to represent its class.

The features for each sample, which are used to train the model, were stored in a separate dataset called OceanProximityPreparedCleanAttributes.csv. In this dataset, the information was prepossessed as mentioned in the assignment description, where each column represents a one of the following features:

1. **longitude** The longitude coordinate of the dwelling's location
2. **latitude** The latitude coordinate of the dwelling's location
3. **housing_median_age** Median age of the block, the lower the number the newer the building.
4. **total_rooms** Total number of rooms within a block
5. **total_bedrooms** Total number of bedrooms within a block
6. **population** Total population of the dwelling's block
7. **households** Total number of groups of people residing within a home unit, per block
8. **median_income** Median income for households within a block of houses (measured in tens of thousands of US Dollars)
9. **median_house_value** Median house value for households within a block (measured in US Dollars)

After analyzing the dataset and identifying the task to perform, we continue to the design process of the model for the feedforward neural network (FFNN) and the its further training.

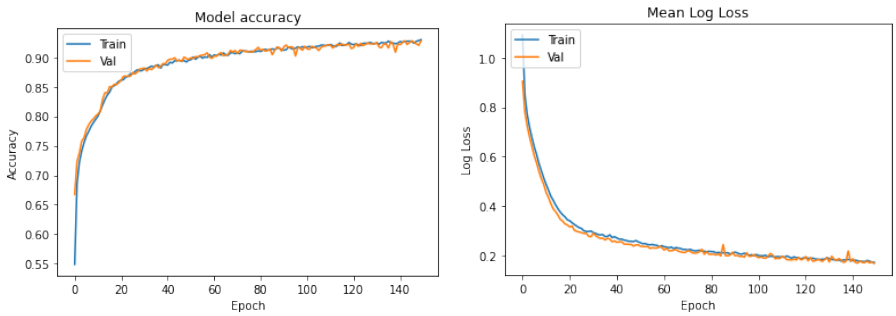
2 Design process

In the design process of the machine learning model performed, we took an iterative approach, where each network was tested after its design only to be modified and tuned in order to get better results in the accuracy over both the training set and the test set.

2.1 Starting model

We started with a simple network which consisted of two hidden layers. The first layer consisted of 1000 neurons, and the second having 750. Both layers where configured with a dropout rate of 0.3 and utilised the Rectified Linear Unit (ReLU) activation function. These layers were followed by a output layer that used the softmax activation function.

The optimizer applied was the Stochastic Gradient Descent, configured to use a learning rate of 0.0001, decay = 1e-6 momentum=0.99 and Nesterov=True and loss = categorical_crossentropy metrics=accuracy. This model was trained over 150 epochs with a batch size of 25. It was able to obtain a test loss of 0.160 and an accuracy of 0.939, these can be seen in Figure 1.



(a) Accuracy (b) Loss

Figure 1: Starting Model Results

2.2 Number of epochs

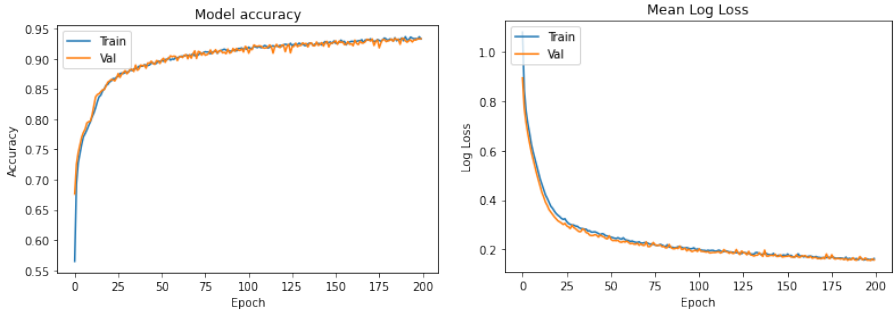
In an attempt to improve on these results the number of epochs was increased to 200, while keeping the batch size at 25. This gave slightly improved results, a test loss of 0.1498 and an accuracy of 0.945 as seen in Figure 2.

2.3 Activation function

To compare the difference between the effectiveness of different activation functions ReLU was exchanged for the hyperbolic tangent activation function (tahn). This change lead to a worse results with a test Loss of 0.255 and an Accuracy of 0.890 as seen in Figure 3. The LeakyReLu activation function was also tested, but the results did not improve either. Therefore, it was decided to use the ReLU going forward.

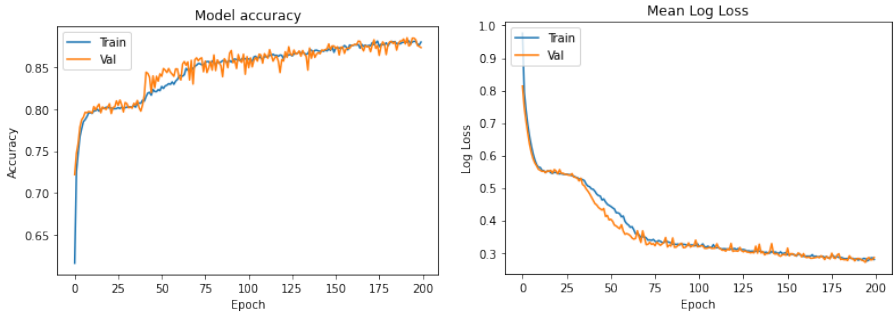
2.4 Number of layers

Realising that the results would not improve greatly through just the modifications of parameters we looked to extend the base model. Thus for the second iteration of the



(a) Accuracy (b) Loss

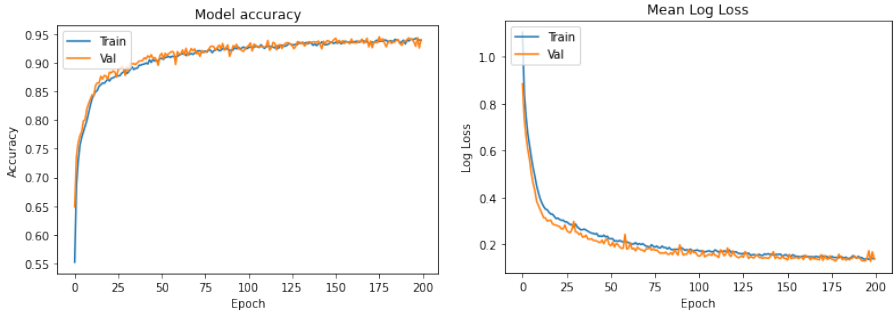
Figure 2: Starting Model with increased epoch count results



(a) Accuracy (b) Loss

Figure 3: Starting Model with tahn activation results

model we began by adding a third layer made up of 250 neurons. This was configured to use same dropout rate (0.3) and ReLU activation function as the previous layers. This addition lead to results a loss 0.124 and and accuracy of 0.952 as seen in Figure 4.

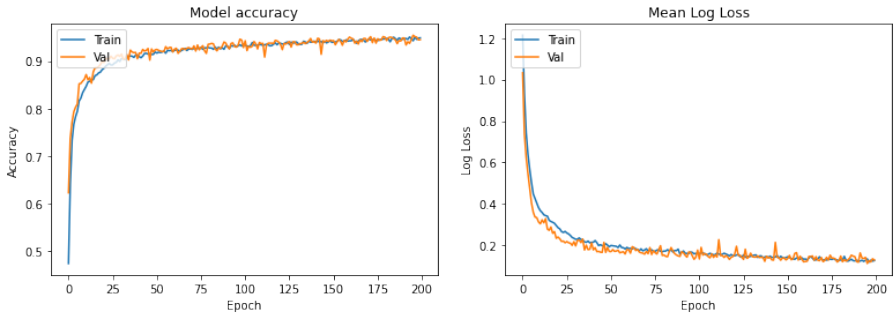


(a) Accuracy (b) Loss

Figure 4: Second Model results

After experimenting with adding additional layers to the model no great improvement in

accuracy was seen, in fact it was seen to drop. This is shown by the example of a model made up of five hidden layers. Starting with 1000 neurons, followed by 750, 500, 200 and finally 100. Each with a dropout rate of 0.3 and the ReLU function. The output layer and optimization parameters being the same as the previous models. With the same number of epochs and batch size it achieved a better loss with 0.113 but a worse accuracy of 0.946 as seen in Figure 5.

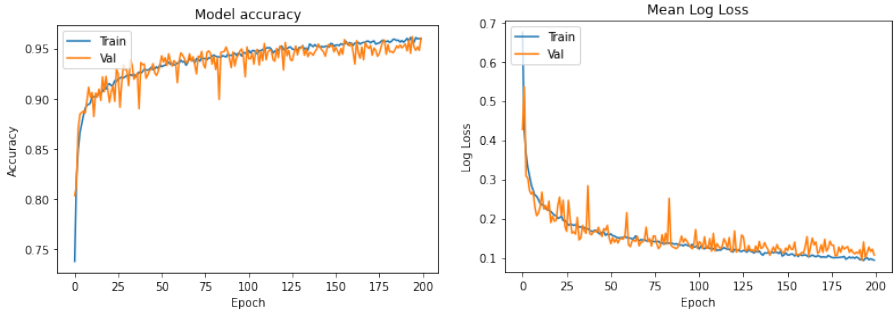


(a) Accuracy (b) Loss

Figure 5: Five layer Model results

2.5 Optimizer

Thus, we investigated if modifying the hyperparameters of the optimizer would improve the results. To do this we modified the learning rate from 0.0001 to 0.01, and the momentum from 0.99 to 0.9. This change in hyperparameters lead to an improvement with a loss of 0.0984 and an accuracy of 0.960 as seen in Figure 6. Although the effect of increasing the learning rate can be seen with the validation lines extreme fluctuation compared to the previous models.



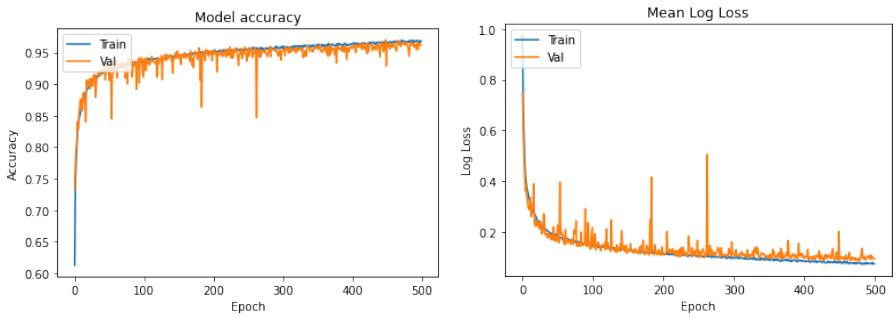
(a) Accuracy (b) Loss

Figure 6: Second Model with modified optimizer results

The optimizers Adam and Adgrad were also experimented with. However SGD outperformed them both so was used for the rest of the development process.

2.6 Batch size and number of epochs

This modest improvement was further advanced by simultaneously changing the batch size from 25 to 128 and the number epochs from 200 to 500. This modification, allowed us to obtained results a test loss of 0.082 and accuracy of 0.973 as seen in Figure 7.



(a) Accuracy (b) Loss

Figure 7: Second Model with increased epoch count and batch size results

3 Final Model and Results

Finally we decided to change the design from scratch, but taking into account what we had learned from the previous iterations. We designed a new FFNN again with 3 hidden layers, keeping the dropout of 0.3 for each layer and the activation function kept to the ReLU function. The difference in the model is that the number of neurons for each layer were changed to 512, 256 and 128. These were followed by an output layer with the same settings as previous, a neuron for each class with softmax activation. A diagram of the network can be seen in Figure 8.

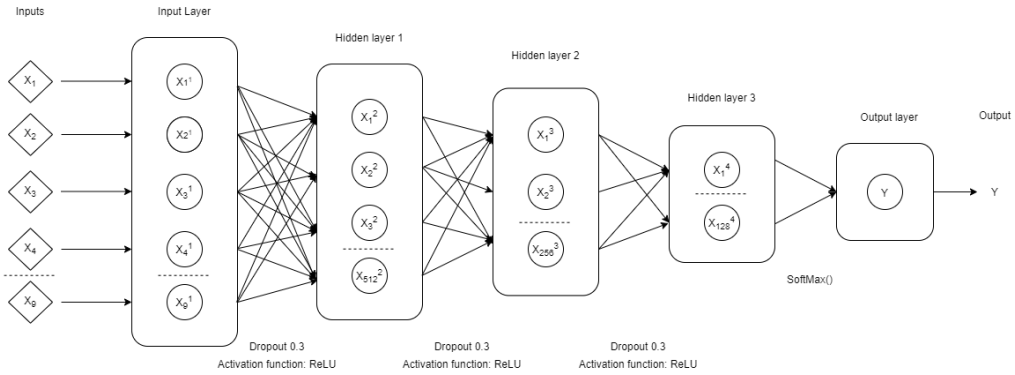
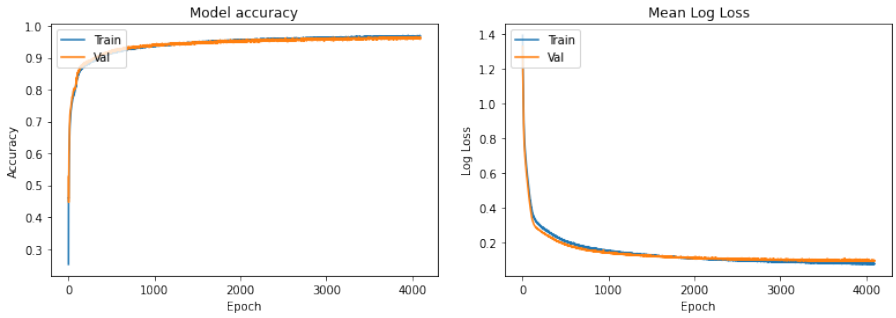


Figure 8: Architecture of final model

The optimizer hyperparameters were set to; learning rate=0.0001 the momentum equals

to 0.90 and kept with Nesterov set to true. Having experimented with different epochs and batch sizes the best results where found to be a batch size of 256 ran over 4096 epochs. This lead to a result of test loss of 0.091 and an accuracy of 0.977, as seen in Figure 9.



(a) Accuracy (b) Loss

Figure 9: Final Model Results

As can be seen from the loss graph this model it is neither under or over fitting as both the training and validation lines are very close together. Although they have not yet flattened out it is unlikely more epochs would improve this model. This can be determined as the two lines are starting to separate in the loss plot with the validation values not improving at the same rate. With this separation there is a threat of overfitting occurring.

3.1 Further modifications

Based on the final model, we tried other modifications to improve the results. Although none of them achieved this objective, we still consider interesting to describe here.

3.1.1 Batch normalization

In an attempt to further improve the performance batch normalization was added to each hidden layer. With this applied the final model achieved an accuracy of 0.904 and a loss of 0.330.

3.1.2 Initializer

We also tried using a kernel initializer. The one tried has been the He Uniform initializer. The final model has achieved an accuracy of 0.975 and a loss of 0.084.

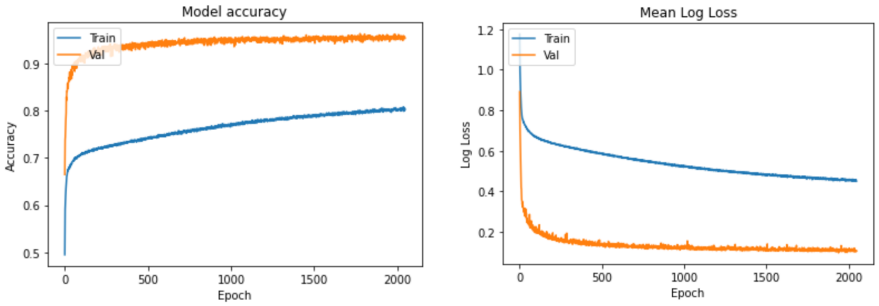
3.1.3 Data augmentation

Since we were having problems further improving the results, we decided to perform some data augmentation. That is, we took the training dataset, created a new one based on

its characteristics and combined both. This way, the network is trained with a larger sample and is exposed to more varied instances. If the process is done correctly, the model is better suited to classify the testing dataset. The augmented training dataset was obtained in the following steps:

1. Divide the training dataset instances based on the class.
2. For each class, obtain the mean and standard deviation of each of the attributes.
3. For each class, generate a new dataset. Each attribute is obtained drawing random samples from a Gaussian distribution with the mean and standard deviation obtained in the previous step.
4. Join the datasets of the different classes together.
5. Create a label dataset in the One-hot encoded fashion that matches the new attribute dataset.
6. Combine the old and new attribute and label datasets and shuffle them randomly, with the same order for both datasets.

The final model obtained an accuracy of 0.961 and a loss of 0.098 with the augmented training dataset. It can be seen in the Figure 10 that there is quite a difference between the training and validation metrics. This happens because the variety introduced in the training dataset makes it more challenging for the network, whereas the validation dataset remains easier to classify. Also, we must mention that the model was trained with only 2048 epochs instead that with 4096.This was done because with a large number of epochs, Google Colab crashed.



(a) Accuracy (b) Loss

Figure 10: Augmented Dataset Results

3.1.4 Number of layers

Finally, we decided to try modifying the number of layers, by increasing and decreasing, improved the results. First, we tried an architecture without the last hidden layer (that

is, with one 512 and one of 256). It got an accuracy of 0.972 and a loss of 0.075. Second, we tried an architecture with five hidden layers (2048, 1024, 512, 256, 128). Its accuracy was of 0.968 and its loss, of 0.145. As can be seen these are both lower than the final accuracy than the final proposed model.

4 Conclusions

Through out this project what has become clear is designing deep-neural-networks is not an exact science, but an iterative process of discovery. As networks are black boxes we are required to make educated guesses on what is required to improve results. What has defiantly been shown is adding additional layers does not lead to these results. During the project our initial response to how we should increase accuracy was more hidden layers although this was quickly shown not to be the optimal approach. As well the accuracy reduction that was commonly saw the other issue was it could lead to far larger building times for no improvement. The actual best changes for results are the configurable parts of a layer such as the number of neurons or the activation function. Then by setting hyper parameters to get the best results from these base models. By performing regularisation through setting a dropout rate we were able to reduce the prevalence of overfitting. While be configuring the optimizer parameters were able to smooth the learning rate of the model. Overall we are happy with our final model. Although it does not have a perfect prediction rate it well within being acceptable as a reliable model.