CO395 - Machine Learning Coursework 2

Padmanaba Srinivasan
01191525

Keerthanen Ravichandran
01195170

Harrison Ankers
01211208

Tze Hei Tang
01221240

March 1, 2019

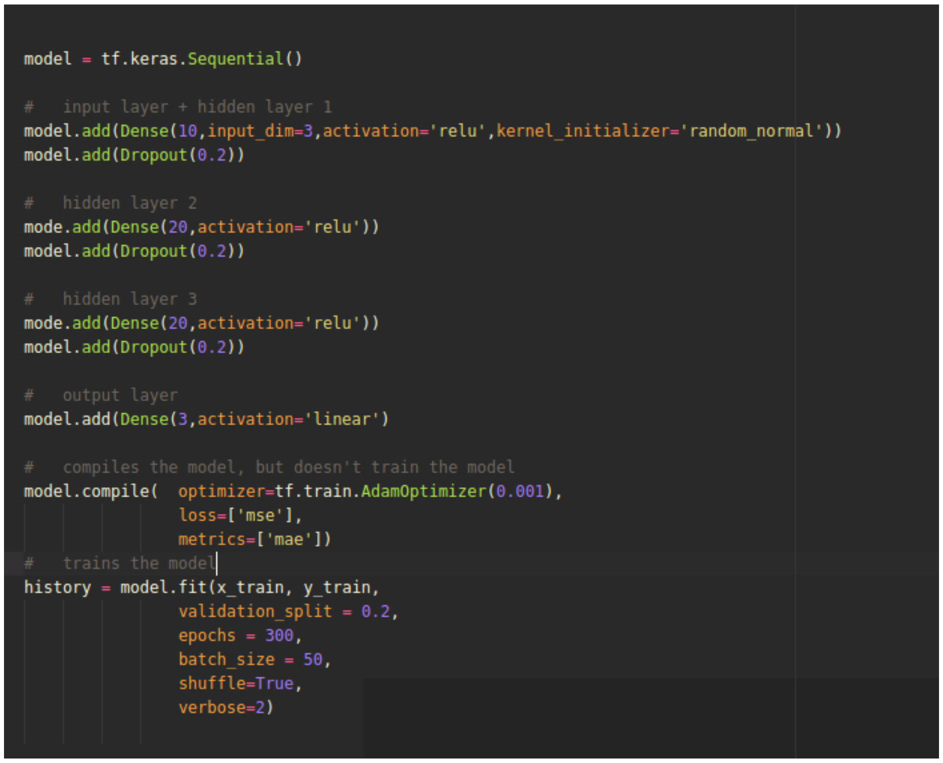
Contents

1 Part 2

1.1 Q2.1: Implementing the forward model of the ABB IRB 120

1.1.1 Creating the Neural Network model using Keras

Using Keras to create a Neural Network Model is trivial and can be accomplished in a few lines of code as in figure 1.



```
model = tf.keras.Sequential()

#   input layer + hidden layer 1
model.add(Dense(10,input_dim=3,activation='relu',kernel_initializer='random_normal'))
model.add(Dropout(0.2))

#   hidden layer 2
mode.add(Dense(20,activation='relu'))
model.add(Dropout(0.2))

#   hidden layer 3
mode.add(Dense(20,activation='relu'))
model.add(Dropout(0.2))

#   output layer
model.add(Dense(3,activation='linear')

#   compiles the model, but doesn't train the model
model.compile(  optimizer=tf.train.AdamOptimizer(0.001),
                loss=['mse'],
                metrics=['mae'])
#   trains the model
history = model.fit(x_train, y_train,
                validation_split = 0.2,
                epochs = 300,
                batch_size = 50,
                shuffle=True,
                verbose=2)
```

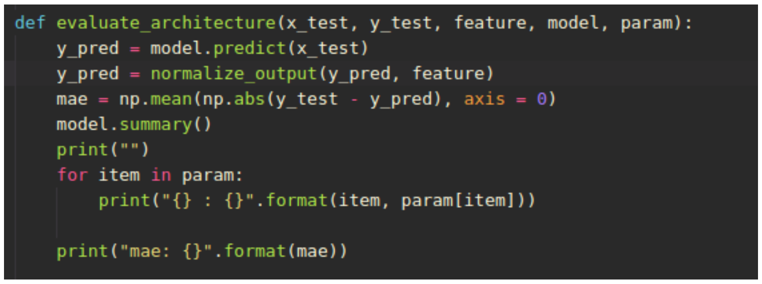
Figure 1: Basic neural network using Keras

This model is created with 3 hidden layers, each with 10, 20, and 20 neurons respectively. All layers use *Relu* as the neuron activation function, with a dropout rate of 0.2. There are three input nodes and three linear output nodes. All the nodes are initialised using a Random Normal method.

An adaptive learning rate method called *Adam* has been implemented with an input rate of 0.001; this method was chosen as it is computationally efficient (which becomes more relevant when optimising hyperparameters) and memory efficient. The Loss Function and Validation Metric are Mean Squared Error (MSE) and Mean Absolute Error (MAE) respectively. MSE and MAE were chosen as the problem is one of regression and these are well suited to this. Furthermore, MAE is more robust to outliers than MSE and using this yielded some improvement in the Neural Network's performance.

Finally, a validation split of 20% was decided upon along with 300 Epochs to ensure there are sufficient number of trials for the network to learn and come to a stable point where further training yields no more benefit. A batch size of 50 was used. The metrics were determined by looking at how regression problems with Neural Networks are usually solved and with intuition about the scope of the problem.

1.2 Q2.2: Evaluating the architecture



```
def evaluate_architecture(x_test, y_test, feature, model, param):
    y_pred = model.predict(x_test)
    y_pred = normalize_output(y_pred, feature)
    mae = np.mean(np.abs(y_test - y_pred), axis = 0)
    model.summary()
    print("")
    for item in param:
        print("{} : {}".format(item, param[item]))

    print("mae: {}".format(mae))
```

Figure 2: Evaluate architecture

The *evaluate_architecture* function (figure 2) outputs the model structure (ie. number of layers and neurons etc.), the hyperparameters chosen for the model, and the accuracy on the test dataset.

1.3 Q2.3: Fine tuning the architecture

1.3.1 Data preparation

The initial dataset was split into a training set, validation set and test set in a ratio of 6/2/2. The entire dataset is first shuffled to ensure that we are not segmenting a part of data that may augment the statistical properties of the data, since the initial dataset is sorted. 20% of the shuffled set is separated to become the test set. This is kept separate from the rest and cannot be used in training the network.

The data would also need to be processed before being fed into the neural net. θ_1, θ_2, and θ_3 have a range of $[-1.57, +1.57]$, which is actually $-\frac{\pi}{2}$ and $\frac{\pi}{2}$. The 3 output corresponds to the Cartesian coordinates of the tip of the hand, x and y have a range of $[-714, 714]$. z has a range of $[-84, 1004]$. This difference in range can be problematic for the back propagation algorithm. For example, if x and y are mostly accurate in an epoch, but z has a slightly higher percentage error, even if that percentage is low, it will have a higher loss value than the loss produced by x or y. As a result the input and output of the network must be normalised. Both the input and output are normalised to the range of $[0, 1]$.

1.3.2 Hyperparameters

In order to fine tune the neural network, a set of optimal hyperparameters must be selected in order to train a model that will gives an optimal result. Hyperparameters are parameters that are not changed throughout the training period (learning rate optimiser *Adam* implements adaptive learning rates, but we still pass a learning rate hyperparameter to it), the tuneable hyperparameters we chose were :

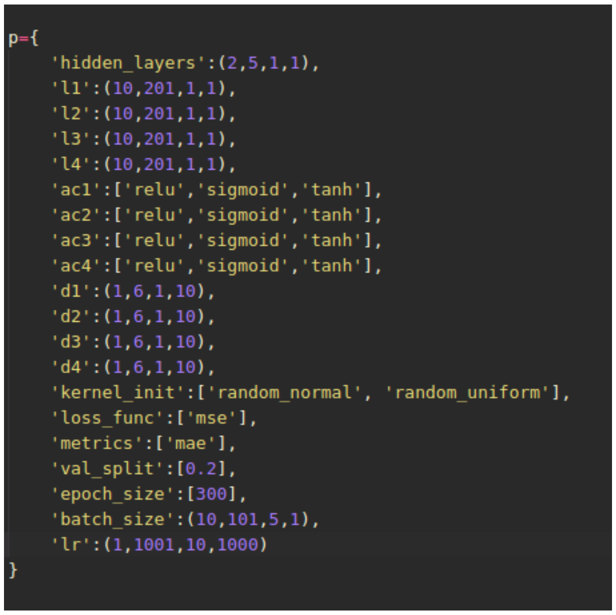
1. Number of hidden layers.
2. Number of neurons in each layer
3. Activation function of each layer
4. Batch size

1.3.3 Methods of optimising hyperparameters

Hyperparameters can be searched for manually. However, as the number of parameters and their values increase the number of permutations increase too and it becomes impractical to do manual search. The most notable methods of hyperparameter search are Grid Search, Random Search, and Bayesian Optimisation.

Grid Search is easy to implement in practice, the hyperparameters are simply iterated through the user-defined set - it takes a brute force approach to optimisation.
Random Search selects a permutation of the hyperparameters randomly. In general Random Search performes competitively but with the added advantage of taking less time as not all combinations are tested. Random Search was used as the method of choice for part two of the task.
Bayesian Optimisation is more complicated and was impractical to implement in the period of this coursework.

1.3.4 Selection of hyperparameters and their range



```
p={
    'hidden_layers':(2,5,1,1),
    'l1':(10,201,1,1),
    'l2':(10,201,1,1),
    'l3':(10,201,1,1),
    'l4':(10,201,1,1),
    'ac1':['relu','sigmoid','tanh'],
    'ac2':['relu','sigmoid','tanh'],
    'ac3':['relu','sigmoid','tanh'],
    'ac4':['relu','sigmoid','tanh'],
    'd1':(1,6,1,10),
    'd2':(1,6,1,10),
    'd3':(1,6,1,10),
    'd4':(1,6,1,10),
    'kernel_init':['random_normal', 'random_uniform'],
    'loss_func':['mse'],
    'metrics':['mae'],
    'val_split':[0.2],
    'epoch_size':[300],
    'batch_size':(10,101,5,1),
    'lr':(1,1001,10,1000)
}
```

Figure 3: Hyperparameter search space

The hyperparameter search space includes the parameters shown in 3. Tuples represent a range denoted by the notation $start, end, step, division.Listsrepresentspecificvaluesthataparamatercantakethevalueof.Thefu$

- The hidden layer can be between 2 and 4 layers in length.
- Layer i can have l_i neurons with range $[10, 200]$.
- Activation functions can either be $Relu$, $Sigmoid$, or $Tanh$.
- Dropout rate of layer i have a range of $[0.1, 0.5]$.
- Node initialisation can be either Random Normal or Random Uniform.
- Batch size can have a range of $[10, 200]$.

- Learning rate can have a range of $[0.001, 1]$.

Loss, Metric, Validation Split, and Epoch size were kept constant.

Mean Square Error was chosen as the loss function to reduce loss and punish larger differences between the output and target. Mean Absolute Error was chosen as the metric to monitor progress and since the output is normalised to range $[0, 1]$, it is essentially the proportion of error.

1.3.5 Code Implementation

Libraries such as $Talos$ exist which work with Keras to implement hyperparameter evaluation, however we decided to write the code for this ourselves.

The algorithm has the following stages: a dictionary of the hyperparameters is created with the name of the hyperparameters as the key, and a possible range of values is passed through. All the elements in the dictionary are then iterated through and a random value for the hyperparameters is selected after which this particular permutation is then ready to be fed into the Neural Network.

1.3.6 Search for optimal hyperparmeters

As Random Search is used, the number of permutations tried by the algorithm is user decided. To search for the optimal solution, we explored a large number of permutations, and selected a few of the best and further evaluated them. The naive approach to run the algorithm n times was impractical as for large n, the program consumes a lot of memory and causes system-wide performance degradation. To circumvent this a bash script was used instead to run call the Python program n times thereby periodically frees up memory. This procedure was left running to produce results for 1000 permutations.

Once a large set of results was obtained, it was sorted according to the validation metric and pass the top few results for further evaluation using k-fold cross validation.

The model with the optimised hyperparameters are saved into a csv file and sorted by its validation metric (the Mean Absolute Error). We can then used test dataset to evaluate and ensure that the performance is still optimal. the test dataset is only used for final evaluation and not used for selection in parameter search. Finally, the parameters, model architecture, and weights are saved to be used afterwards.