INNS Report **Exam No: Y3868349**

Discussion of Architecture

Overview of the problem

This is a regression problem. Because there is an unlimited number of possible outputs and only one output is required. The output is based on the target 'rented bike count' column which varies depending on the other columns which can be seen as inputs. The problem is to predict how many bikes are needed at a certain time (based in Seoul) depending on the weather/time values inputted. So, given previously unseen data, the neural network model should be able estimate accurately the number of bikes needed at that point in time. There are 12 input columns including 'Hour' of day and all the weather columns such as 'Temperature' for example. The 'Date' column is not to be used as it provides no real value to the model. Its data type is also not usable, and it doesn't help predict future data. Of the 12 input columns, 3 columns are categorical data types whilst the others are continuous. However, the categorical columns are also quite useful as they correlate well with the output column. So, in that case it would be a good decision to use dummy variables or original encoding (depends if ordinal or nominal data) to translate them into a numerical format so that they can be fed into the neural network. Overall, there are 8760 data points. This is a large number of possible data points so it would allow a flexible ratio in terms of the train/test/validation split. A large number of data points will definitely help training accuracy and the model's ability to predict new values as a whole however I must take caution with training times due to the size of the dataset.

Review of types of neural networks

A normal feed forward network is unsuitable for this dataset. It has no cycles so can't learn from previous inputs. This would be crucial for this problem as we are using a very large dataset. And so, it follows that perceptron's are also not useful as this is not a binary classification problem.

An autoencoder will also not work on this problem as it is designed to reproduce the same input as the output (albeit in lesser size). However, the problem is simply not suited to this architecture as this is a regression problem and not a compression/de-noising/feature extraction problem.

Convolutional Neural Networks are a possibility however, they are mostly used for image classification, time series analysis and Natural Language Processing. Even Though a CNN can be adapted to work with regression it may not be useful in this instance as the input data is from a CSV file and is not an image/image-based data set. CNNs are also set to have a higher computational cost.

A Radial basis functions network can be used for regression as well [4]. However, it only has one hidden layer so it will limit my capability in terms of combatting overfitting through amending the structure of the architecture.

Initial exploration of data

The dataset is quite good in the fact that it doesn't have any empty or missing values. The data seems unbalanced however with higher values being underrepresented, I will explore this and other outliers (such as 0 rented bikes) later. Dummy variables can be used for the 'Seasons' column as the variables aren't necessarily 'higher' or 'lower' than each other so it makes sense to separate them out and see when each season is selected; this would be identified as a '1' in each seasons' column. So only one season column at any time would have a '1' and the rest would be 0s in the same row (data point).

Selected Architecture

A Recurrent Neural Network with LSTM seems to be the best option for this problem. One main factor to consider is that the data set had data points in order of time from earliest to latest. And data at same times have similar number of rental bikes required and there is a clear correlation between different times of day. So, an LSTM would be useful as it would be able to use previous data points in its prediction for a certain data point. RNN's suffer from not being able to handle long term dependencies: i.e. when the gap between the input where the relevant information is, and the current time step is too large. LSTM's don't suffer from this issue as they are designed to remember information for a long time. It has built in layers that optionally chooses to remember/forget. So, for the given dataset this is important as depending on past seasons and times there are patterns. Therefore, remembering these patterns is crucial. It's layers and neurons can also be easily manipulated through Keras so for ease of use is also a valid reason.

Creation and application of neural networks – (Using Python/Keras)

I will be using **Mean Squared Error** to calculate the loss (error) in my network. And **R Squared/Coefficient determination** to calculate the accuracy. This will be explained further in the results section but from this point onwards I will refer to them as just 'loss' and 'accuracy' for ease.

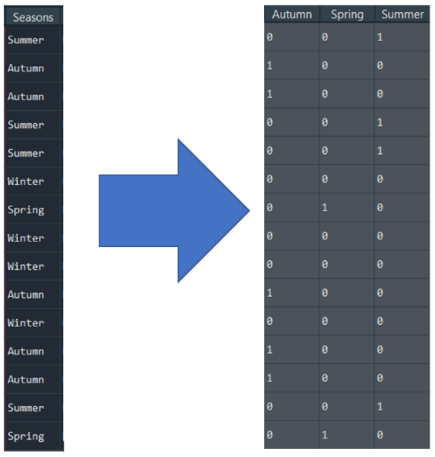
Discussion of features – Initial Chosen Inputs (All apart from date)

(1) Hour: An important stat as the amount of bike required varies highly with the time of day (measured at each hour). **(2) Temperature, Humidity, Wind Speed, Visibility, Dew Point Temp, Solar Radiation, Rainfall, Snowfall:** The weather values are all likely to correlate slightly with the number of bikes needed at any point in time. However, some will have more of an effect than others. It would be a good starting point to use all and then later if further accuracy is needed; a number of columns can be dropped from the model. **(3) Date:** The column date is not chosen as it does not bear any effect on the output. **(4) Seasons:** Various seasons are likely to have more or less demand for rental bikes. From looking at the data it looks like demand is higher in the warmer seasons such as Spring and Summer and less so in Autumn and Winter. However dummy variables will be used to encode this categorical data. **(5) Holiday:** Will need to use original encoding as there are only two options so is a binary encoding. It is likely to have high correlation with the output as it is quite a big change in circumstances for students/employees as they will not travel as much during holidays. **(6) Functioning Day:** Bike rent count will always be 0 on a non-functioning day. So, might be worth deleting this from the data later to improve accuracy on functioning days. It does not provide any use to the problem as a NN model is not needed to know whether a day is a functioning day or not (and therefore whether 0 bikes are needed).

Data Pre-processing - Initial

There are no missing values in the dataset so we can be confident that the dataset won't lead to errors when fed into the model.

I will next randomize the order of rows in the data set; in other words, the data points will appear in random order. This will allow us to gain a more representative train, validation, and test splits. So, the model should be able to gain a good accuracy for most importantly any unseen data. And also, the unseen data being fed into the model will not have to be in chronological order. Next, I decided to convert the Seasons column like so:



Seasons
Summer
Autumn
Autumn
Summer
Summer
Winter
Spring
Winter
Winter
Autumn
Winter
Autumn
Autumn
Summer
Spring

Autumn	Spring	Summer
0	0	1
1	0	0
1	0	0
0	0	1
0	0	1
0	0	0
0	1	0
0	0	0
0	0	0
1	0	0
0	0	0
1	0	0
1	0	0
0	0	1
0	1	0

I used the dummy variables concept to make this change [1]. So, each variable in the 'Summer' column is converted into its own column. A '1' represents that season is the current season. So only one of the three seasons can have a '1' at any row. The rest will have zeroes. I decided to remove 'Winter' due to the fact that keeping all 4 variables increases **multicollinearity** in the dataset [1].

Next, I will use original encoding for the other two columns: 'Holiday' and 'Functional Day'. Because both only have two variables and can be represented easily as binary inputs. However, this isn't always the most optimal method so I will try the previous method for these columns as well if accuracy isn't high enough during testing of the model on unseen data.

Normalizing the data is the next task. Neural networks work better when all the values including in the data column are between 0 and 1.

Finally, the dataset has to be split into train, validation, and test sets. To start off with I will split them into a 64:16:20 ratio.

Initial Model Selection

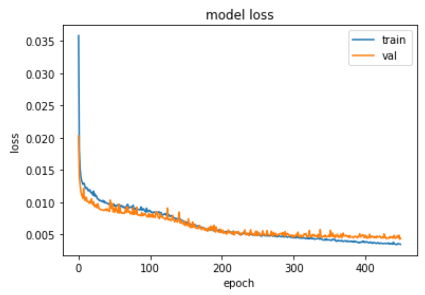
I needed an initial model to first test the model. It is extremely unlikely that I will choose the most optimal hyper parameters at this stage. But by being reasonable with my choices I can get a good enough start and then change parameters accordingly in an attempt to increase accuracy and minimize loss.

Neurons-Hidden Layers	**Activation-Hidden Layers**	**Optimization Algorithm**	**Learning Rate**	**Batch Size**	**Number of hidden layers**
50	Tanh	Adam	0.001-Default	64	4

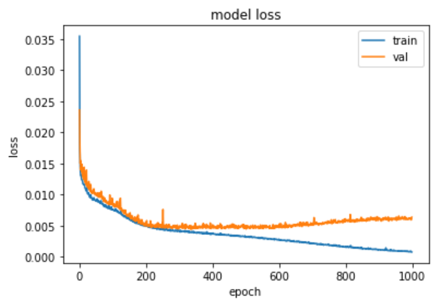
Results on Test Data:

Epochs	**Loss**	**R-Squared**
150	• 0.00693129887804389 • 0.006647439207881689 • 0.004979646299034357	• 0.7997003197669983 • 0.7895340919494629 • 0.8208362460136414
300	• 0.0051879361271858215 • 0.004173579625785351 • 0.004873545374721289	• 0.820329487323761 • 0.8644331693649292 • 0.8373813033103943
450	• 0.004187928047031164 • 0.0049284156411886215 • 0.004215745721012354	• 0.8547372221946716 • 0.8497108221054077 • 0.8555055856704712

From the above data it is clear to me that the average highest accuracy is achieved with 450 epochs. However even though it has the highest accuracy out of the three epochs tested, it shows signs of slight overfitting as the training loss/accuracy stayed steadily better than the validation loss/accuracy towards the last quarter of the training process:



So, I can infer from this that the full performance of this model has not yet been reached. I think at this stage adding dropout to the model (in the hidden layers) would be the logical step forward. As this would prevent overfitting to the training data at every epoch as some random neurons are switched off/ignored so all of the training data is not always trained on. Dropout is a form of regularization so should allow us to prevent this overfitting. Before adding dropout, I ran a 1000 epochs test of the same system to see if I can learn more about its characteristics:



Even though it achieved an accuracy of 0.97 on the training data it only achieved 0.81 on the test data. So, it is clear to see that this model is prone to overfitting. And we can estimate that the overfitting starts at around 220 epochs on average. Depending on how the data has been shuffled and model has performed this number could vary slightly.

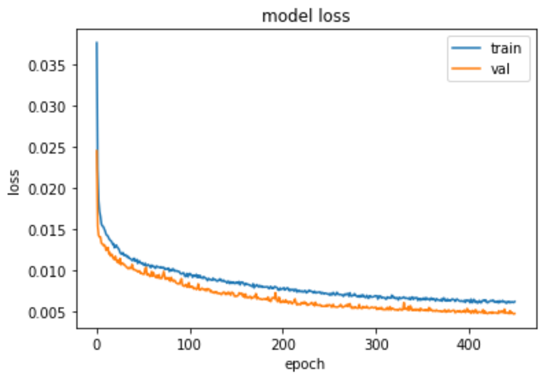
Next, I tested the model with dropout regularization as mentioned. I tested with 0.1, 0.2 and 0.3 dropout. All on 450 epochs (As it produced the best results without dropout) and no other changes to the system was made.

Results on test data

Dropout	**Loss**	**R-Squared**

0.1	• 0.004735312890261412 • 0.004250839352607727 • 0.005039363168179989	• 0.8458797335624695 • 0.8630897998809814 • 0.846966564655304
0.2	• 0.005471535492688417 • 0.004983417689800262 • 0.0052245785482227	• 0.8395458459854126 • 0.8424233198165894 • 0.8319516181945801
0.3	• 0.006135831121355295 • 0.005598207470029592 • 0.005194034893065691	• 0.8184673190116882 • 0.8162705898284912 • 0.8287806510925293

It is clear from these results that Dropout with 0.1 is more optimal than no dropout, dropout with 0.2 and dropout with 0.3. Both 0.2 and 0.3 had quite interesting epochs/loss graphs: the validation loss seemed to be steadily minutely lower than training loss:



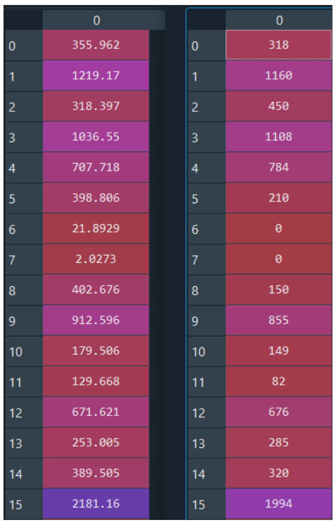
Dropout 0.3 Example

This could be caused by a higher dropout as dropout is not usually activated on the validation set.

On the other hand, for the dropout with 0.1 model the training and validation loss were very similar overall and on toward the last 50 epochs did the model very slightly start to overfit (shown by training loss lowering ahead of validation loss). However, considering all this it seems to be the best model so far; with it comfortably outperforming the others in terms of loss and R squared.

Current performance and additional steps to maximise model performance

My best accuracy so far is around 0.86. For a model that is used to predict the number of bikes that would need to available at a certain time, this isn't too bad. For more important use cases like medical or health related data problems then we definitely need more accuracy. However, it would still be useful to work on the model more as a higher accuracy would mean less energy and manpower is wasted transporting and delivering these bikes around.



	0		0
0	355.962	0	318
1	1219.17	1	1160
2	318.397	2	450
3	1036.55	3	1108
4	707.718	4	784
5	398.806	5	210
6	21.8929	6	0
7	2.0273	7	0
8	402.676	8	150
9	912.596	9	855
10	179.506	10	149
11	129.668	11	82
12	671.621	12	676
13	253.005	13	285
14	389.505	14	320
15	2181.16	15	1994

On the right is a short extract of test results for my best model (dropout 0.1). It has been trained then tested on unseen data with an accuracy score of 0.8690. The RHS column shows the actual number of bikes required and the LHS column shows the number of bikes my model predicted.

As you can see overall the model is accurate enough. However, it seems to underperform when smaller predictions are needed. When the model needs to predict < 50 bikes it seems to underperform, especially when the prediction should be 0. The same holds for extremely high values as well.

Therefore, the next logical step for me seems to be to balance the dataset further. There are 295 datapoints where the required number of bikes is 0. These ALL occur when 'Functioning Day' is 'No'. Because the firm is closed on a non-functioning day this makes a lot of sense. I don't think a neural network is therefore needed to predict the number of bikes required on such days. This should help balance the dataset a bit more and lessen the effect of outliers. So, the 295 rows and the column 'Functioning Day' shall be removed. Below is the test for this with no other changes made.

Results with 'Functioning Day' removed and rows with 0 bikes required:

Epochs	**Loss**	**Accuracy**
450	• 0.004486676305532455 • 0.004230608402192593 • 0.004406670020520687	• 0.8461594581604004 • 0.8618257402420044 • 0.8639933753013611

There seems to be a slight increase in performance after this change has been made. This makes sense as there will be less error due to less 0 values to predict. And this means the model will not use these rows incorrectly to make unreliable correlations between columns as the only column that actually mattered for 0 valued rows was the 'Functioning Day' column.

Further Improvements

Next, I can try a different number of neurons/layers to see if I can improve the accuracy further. This is because, a deeper network could allow for more precision or a shallower network may also work better. I am currently using the **Adam** training algorithm with a learning rate of 0.001. However perhaps better accuracy can be achieved by using a lower or higher rate.

Optimizer/Training Algorithm

I chose the Adam optimizer initially as it is in widespread use in modern neural networks in terms of deep learning. Since I am building a deep model with a large dataset, I decided to test Adam first. However now I will also try the others. SGD is perhaps too limited (especially in terms of speed) for this problem but there are others such as RMSprop. Coupled with different activation functions the results could differ. So first a quick test to settle on the optimal training algorithm and activation function for the hidden layers. Then I will employ random search to finalize on the structural parameters: number of hidden layers, number of neurons in each layer and learning rate.

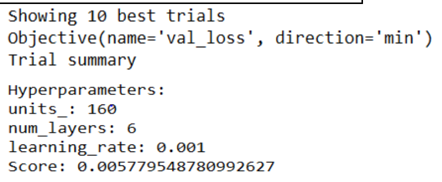
Optimizer	**Hidden layer activation**	**Loss**	**Accuracy**
RMSprop	relu	0.004885236267000437	0.8377885580536886
Adam	relu	0.005171118304133415	0.8341823220252991
RMSprop	tanh	0.004237176850438118	0.8604499101638794

It seems that RMSprop with tanh is the stronger option at this stage. After running multiple times, it gave consistently high results and little to no signs of overfitting unlike Adam. The training loss/validation loss graph is also smoother with less fluctuations. This all makes sense as RMSprop is recommended specifically for RNNs even though Adam is more widely used. Therefore, I will neglect Adam here on in and switch to RMSprop.

Random Search

Now that the training algorithm and dropout has been finalized it will make settling on the rest of the hyperparameters easier as there will be a lower search space. This will save computational time. I will be using Random search to see if there may be a set of structural parameters that will yield better results than what I have achieved thus far. The rest of the parameters apart from Number of layers, neurons and learning rate, I kept the same.

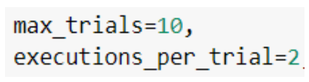
Hyperparameter	**Search Space**
Number of hidden layers	Min: 2, Max: 22, Step: 1
Number of neurons in each layer (excluding output layer)	Min: 32, Max: 256, Step: 32
Learning Rate (RMSprop)	3 Choices: {0.01, 0.001, 0.0001}



```
Showing 10 best trials
Objective(name='val_loss', direction='min')
Trial summary
Hyperparameters:
units_: 160
num_layers: 6
learning_rate: 0.001
Score: 0.005779548780992627
```

The optimal result was 160 neurons per layer, 6 hidden layers and a learning rate of 0.001. Score is validation loss.

After testing these hyperparameters myself afterwards I found that there were no real improvements. MSE stayed at around 0.0048 and R Squared at 0.83. So random search had failed in this instance. This may be due to the settings I chose which is shown.

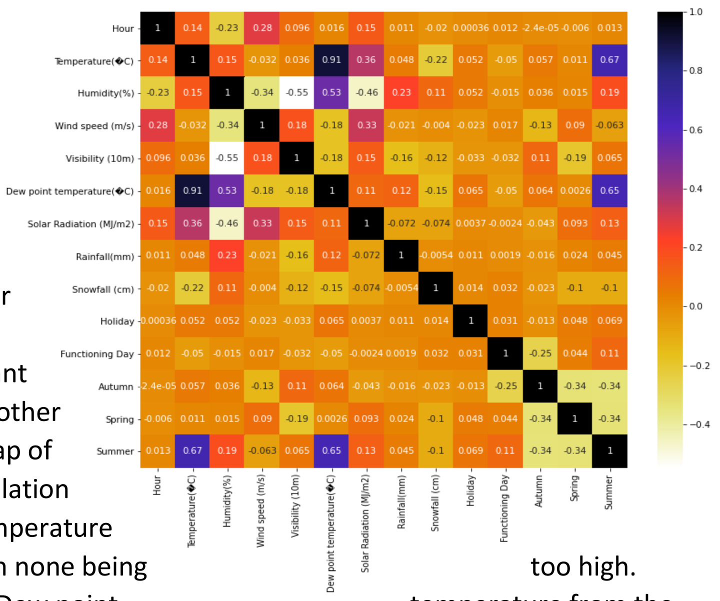


```
max_trials=10,
executions_per_trial=2
```

For the lack of time and scale of the project it didn't make sense for me to run a test over many hours. Especially considering the lack of high-end hardware. Regardless, this test still took 4 hours 6 minutes. The problem with this is that not enough combinations were tested and because I ran each trial for 100 epochs it meant that perhaps out of the tested networks, the network that converges towards the solution the most quickly will do well on this test rather than the actual “best” network. So more stable networks that converge perhaps slower were not rewarded as opposed to the quick but perhaps unstable ones.

Pearson Correlation – Feature selection technique

With this many number of features, it wouldn’t be surprising if several features were highly correlated with each other. Having multiple features highly correlated to each other leads to multicollinearity in the dataset [2]. This means that other features that are also significant have less impact during training when multiple other features are highly correlated. Here is a heat map of the correlation in the dataset. The highest correlation exists between Dew point temperature and Temperature (0.91). Others have mostly fine correlations with none being too high. So next step is to drop Temperature or Dew point temperature from the dataset and test the results:



Dropped Column	Loss	Accuracy
Dew Point Temperature	• 0.0047301920130848885 • 0.00451003760099411	• 0.8410413265228271 • 0.841121256351471
Temperature	• 0.0048882258124649525 • 0.005348438862711191	• 0.8485701680183411 • 0.8323739767074585

No significant change can be seen. Perhaps because we only had to eliminate one column. The change will be kept (eliminate dew point temp) even though the loss seems slightly higher than on the previous runs without the change. This is because we still are able to eliminate multicollinearity so this change may be seen in more unseen data more positively, also the other features will have more impact when training.

Results and Evaluation

Metrics

The main two metrics I used in the whole process is Mean Squared Error and R squared. MSE is useful for this problem as it determines the average difference between the target y value and the calculated y value. MSE was preferrable over Mean Absolute Error as MSE punishes outliers more. Since the data has a handful of outliers (31 cases where >3000 bikes required) it makes sense to use MSE. R Squared was chosen as it makes it easier to see the “accuracy” of the model. It shows how well the model fits to the data given. But it may fail to punish outliers which has to be considered. For all the tests I used a combination of the two to determine my next decision and usually the two values were well correlated which makes it easier. R squared makes interpreting the MSE easier as well because MSE is affected by scaling of the input data. So, a very small MSE could in fact be misleading in which case R Squared is easier to read as the close to 1 it is the better the model. Both stats were available in for all 3 of test, train, and validation sets. I prioritised the testing set as it was completely unseen. And if values were similar, I fell back on the values for the validation set.

Final Results

Final Model: Hidden Layers (LSTM): 4, Hidden Layer Neurons: 50, Activation Function: tanh, Training algorithm: RMSprop, LR: 0.001, Dropout after each hidden layer: 0.1.

Epochs	**MSE**	**R Squared**
450 – (5 Run average)	• 0.004279468882083893	• 0.8611045589447021

(Results after "Functioning Day" and "Dew point temp" was removed from the data.

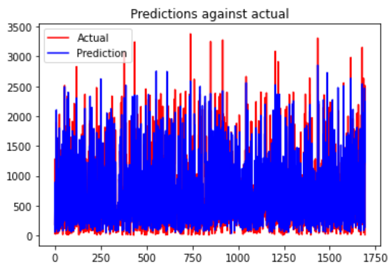
Evaluation

The model tops off at about 0.86 R Squared and 0.0042 MSE in the final test. This hasn't changed from the initial model despite my best efforts. However, the probability of this optimal value occurring has definitely increased. Because this machine isn't to be used in a more serious scenario where near to perfection is required, I think it can be used for the bike prediction use case it was designed for. I can mostly deliver accurate predictions =-100. Its weakness lies in extremely low or high target values as it underestimates these. But for normal days the model should be able to be used in the real-life situation. The model could have more performance in it; I could've implemented random search with variable number of neurons per layer and dropout, but time was an issue. And also, could've trialled more learning rates. However, with all the changes made to the dataset and trialling all possible architecture layouts, then I'm confident that with a higher number of trials; random search could've produced a more optimal configuration. Limiting the search space was the reason I wasn't able to make proper steps forward in performance. However, the performance I was getting was accurate enough for real world use as its benefits would mean less people and manual energy would need to be put in to estimate number of bikes needed. Perhaps it can be used as a tool for assistance in estimating rather than a sole predictor.

Something that is quite telling is the difference in performance between different runs. The model sometimes quickly whilst it takes longer on other runs. And the actual MSE/R Squared value differs between runs whether on testing set or validation/training set. This is approx. in the range 0.0040-0.0047 MSE/0.82-0.87 R-Squared. This could mean that the dataset is still quite imbalanced in its spread of values. As the dataset gets randomized for each run so this is likely to be the biggest change between runs of the same network. So, a more complete dataset could help train a better performing model. So, I decided to run a quick experiment to see the range of values:

```
Number of rows < 1000 bikes: 6340
Number of rows < 2000 bikes and >= 1000: 1936
Number of rows < 3000 bikes and >= 2000: 453
Number of rows >= 3000 bikes: 31
```

This shows significant imbalance. This spread of data shows that higher values are being undersampled and lower values are being oversampled. This is in fact proves the weakness of my model created thus far: as you can see it's weakness lays in predicting the higher values. It seems to under-predict these. Sor for my experiment I decided to duplicate all rows whose bike count was >1500. The code was quite simple: I am simply selecting the required dataset then adding it to the original dataset. The results I achieved was fascinating:



```
over_set=df.loc[df['Rented Bike Count'] >= 1500]
df= pd.concat([df,over_set], axis='rows')
```

Epochs	Train MSE	Train R Squared	Validation MSE	Validation R squared	Test MSE	Test R Squared
450	0.0052	0.8781	0.0045	0.8880	0.0043746670708060265	0.8870921730995178

The scores I achieved on the validation and test sets were the "best" so far. I ran it multiple times and the test R Squared score was consistently >=0.88. However the reason I won't include this in my final model is because it is not neccasarily a "fair test". This is because I am duplicating rows that will also end up in the test set. But if I had more time I would've implemented it more skilfully. But this experiment proves that having a balanced dataset is important particularly in this implementation as the model doesn't seem to be managing weights efficiently for an imbalanced dataset. So I can confidently say that oversampling the high values and/or undersampling the lower values will yield significantly better results than the older model.

Further Application

This problem is comparable to the initial problem as we are using sequential data to start off with. However, it differs in that this is a time series model; so previous records are used to compute the output for one individual. And the number of previous records being used is up to the developer's choice and how much of an individual's data is available. But essentially this will be a time series classification model but with perhaps an extra step in pre-processing as previous records will have to be computed into one input multiple times: i.e. once for each time step for each person. This could lead to a very large dataset to be fed into the NN and could mean much longer training times than for my problem even.

I would recommend a LSTM-FCN (CNN + LSTM) [3] model for this problem. An LSTM is good at choosing what information to keep or forget from previous inputs. And in a time series problem this will be crucial in accurately predicting the output. Depending on the size of the data set we can select the number of layers/neurons. And most likely either Adam or RMSprop would be optimal for this model. The LSTM is better than an ordinary RNN as it deals with the vanishing gradient problem which occurs on long time series data. This is when parameters in a hidden layer don't change much at all as the loss decreases. LSTM fixes this issue by not directly multiplying the outputs by the current weight. It makes incremental changes by remembering previous changes "more". Because this particular dataset is measured at 5s intervals it is likely to be very large. .

Three Convolutional layers can be used to extract the time series data simultaneously with the LSTM layers [3]. The advantage of having two different structures in the same model is that the input is perceived in two different ways. This does increase the complexity of the system however it produces state of the art results [3]. If hardware limits aren't a problem, then this system will work exceptionally with lower training times than usual. The outputs of the simultaneous layers (LSTM and CNN Pooling) will be concatenated before the SoftMax output layer [3].

Random search can be used to settle on the hyperparameters. If high end hardware is available, I would suggest testing a very large search space. Finding the optimal system initially would be a huge step forward as less energy will be wasted trying to find optimal parameters manually.

The performance of the model can be measured using categorical accuracy. This would be the percentage of inputs that were predicted correct. Again, it would be useful to compare from train, validation and test sets with test being the most useful as it is unseen data. And if validation score is lower than the train score on average then it means the model is suffering from overfitting in which case the developer can try regularization techniques such as dropout or try changing the learning rate.

Transfer learning can also be used to improve performance. It works by training the pre-trained model further on new datasets [3]. It is a popular concept that is used for large scale projects when training time needs to be saved. For example, for video classification problems, there are a lot of image classification models like GoogLeNet whose last few layers can be re-trained for a different problem/dataset. Same concept can be used here for time series classification to save time and produce optimal results.

References – Bar lecture material

[1]"The Dummy's Guide to Creating Dummy Variables", *towardsdatascience*, 2017. [Online]. Available: https://towardsdatascience.com/the-dummys-guide-to-creating-dummy-variables-f21faddb1d40.

[2]J. Brownlee, "How to Calculate Correlation Between Variables in Python", *Machine Learning Mastery*, 2018. [Online]. Available: https://machinelearningmastery.com/how-to-use-correlation-to-understand-the-relationship-between-variables/.

[3]F. Karim, S. Majumdar, H. Darabi, and S. Chen, "LSTM fully convolutional networks for time series classification," *IEEE Access*, vol. 6, pp. 1662–1669, 2018.

[4]M. Deshpande, "Using Neural Networks for regression: Radial basis function networks," Pythonmachinelearning.pro, 28-Oct-2020. [Online]. Available: https://pythonmachinelearning.pro/using-neural-networks-for-regression-radial-basis-function-networks/.