Project 4: Neural Networks Project

All code was complied and run in Google Colab as Neural models take time to run and the university laptops donot have enough processing power to run the same.

All comments and conclusions have been added right below each code block for easier analysis and understanding

Open in Colab
(https://colab.research.google.com/github/adithyarganesh/CSC591_004_Neural_Nets/blob/main/Final_NN.ipynb)

Task 1. Automatic grid search

Libraries

Key libraries used are keras and scikit-learn

In [8]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import statsmodels.api as sm
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import GridSearchCV, cross_val_score, KFold
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
from keras.wrappers.scikit_learn import KerasRegressor
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import Adam
```

In [2]:
```
data = pd.read_csv("20.csv", header = None)
```

In [3]:
```
data.head()
```

Out[3]:

	0	1	2	3	4	5
0	94.039	142.51	221.27	339.26	340.85	9503.0
1	107.660	170.76	199.92	310.61	332.60	10107.0
2	61.967	143.51	231.63	305.04	328.60	7506.3
3	86.851	107.66	216.18	333.62	320.53	8724.5
4	78.773	148.64	251.70	322.27	346.57	8713.1

In [4]:
```
data.corr()
```

Out[4]:

	0	1	2	3	4	5
0	1.000000	0.014514	-0.006043	-0.017395	0.016438	0.951735
1	0.014514	1.000000	0.032741	-0.001900	-0.037500	0.046233
2	-0.006043	0.032741	1.000000	0.017146	0.031595	0.098727
3	-0.017395	-0.001900	0.017146	1.000000	-0.009506	0.111425
4	0.016438	-0.037500	0.031595	-0.009506	1.000000	0.179275
5	0.951735	0.046233	0.098727	0.111425	0.179275	1.000000

From the correlation values determined for the dataset, we notice that there is a high correlation with the first column in comparison with the rest

Splitting the data into train and test with a 2000 - 300 split

In [5]:
```
dataset = data.values
X = dataset[:,0:5]
Y = dataset[:,5]
X_test = X[-300:]
X = X[:-300]
Y_test = Y[-300:]
Y = Y[:-300]
```

First, I decided to run a baseline model and see how the mse value for it is coming to be as this would give a perspective of how the values can increase with modification and hyperparameter tuning.

In [6]:
```
# define base model
def baseline():
    model = Sequential()
    model.add(Dense(5, input_dim=5, kernel_initializer='normal', activat
ion='relu'))
    model.add(Dense(1, kernel_initializer='normal'))
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model

estimator = KerasRegressor(build_fn=baseline, epochs=100, batch_size=5,
verbose=0)
kfold = KFold(n_splits=10)
results = cross_val_score(estimator, X, Y, cv=kfold)
print("Baseline: %.2f (%.2f) MSE" % (results.mean(), results.std()))
```

```
Baseline: -8204048.46 (24315010.89) MSE
```

As seen above, for a simple multilayer perceptron regressor, a very high mse value has been determined. This allows us to conclude that better hyperparameter tuning is required with modifications to other parameters such as learning rate, dropout, epochs etc.

Initially, I decided to nail down which an ideal optimizer would be, then I decided to tweak the other major parameters as it takes hours to try every combination.

For a list of optimizers, epochs and batch sizes, I was able to conclude that Adam optimizer is the most ideal for the dataset given to me.

The mse values for each combination while run in gridsearch has been listed below.

```
Best: -25979.201172 using {'batch_size': 20, 'epochs': 100, 'optimizer': 'adam'}
-83848.133594 (31485.334665) with: {'batch_size': 10, 'epochs': 10, 'optimizer': 'adam'}
-124149.147656 (106041.321994) with: {'batch_size': 10, 'epochs': 10, 'optimizer': 'RMSprop'}
-17538629.000000 (6145950.034129) with: {'batch_size': 10, 'epochs': 10, 'optimizer': 'Adagrad'}
-28976.654297 (6686.122457) with: {'batch_size': 10, 'epochs': 50, 'optimizer': 'adam'}
-28985.950000 (4135.118675) with: {'batch_size': 10, 'epochs': 50, 'optimizer': 'RMSprop'}
-1475655.350000 (144409.180757) with: {'batch_size': 10, 'epochs': 50, 'optimizer': 'Adagrad'}
-31307.830078 (7195.229146) with: {'batch_size': 10, 'epochs': 100, 'optimizer': 'adam'}
-35668.427344 (12147.983446) with: {'batch_size': 10, 'epochs': 100, 'optimizer': 'RMSprop'}
-1435397.200000 (173003.982770) with: {'batch_size': 10, 'epochs': 100, 'optimizer': 'Adagrad'}
-607021.156250 (225326.076199) with: {'batch_size': 20, 'epochs': 10, 'optimizer': 'adam'}
-155434.096875 (67205.428782) with: {'batch_size': 20, 'epochs': 10, 'optimizer': 'RMSprop'}
-39172515.600000 (7229904.980792) with: {'batch_size': 20, 'epochs': 10, 'optimizer': 'Adagrad'}
-32730.587109 (9326.100937) with: {'batch_size': 20, 'epochs': 50, 'optimizer': 'adam'}
-46073.637109 (17537.055165) with: {'batch_size': 20, 'epochs': 50, 'optimizer': 'RMSprop'}
-1622539.675000 (233324.938891) with: {'batch_size': 20, 'epochs': 50, 'optimizer': 'Adagrad'}
-25979.201172 (3285.793231) with: {'batch_size': 20, 'epochs': 100, 'optimizer': 'adam'}
-44877.579688 (7302.797490) with: {'batch_size': 20, 'epochs': 100, 'optimizer': 'RMSprop'}
-1489904.750000 (215725.142852) with: {'batch_size': 20, 'epochs': 100, 'optimizer': 'Adagrad'}
-1350494.175000 (162489.428364) with: {'batch_size': 40, 'epochs': 10, 'optimizer': 'adam'}
-742374.950000 (163049.310736) with: {'batch_size': 40, 'epochs': 10, 'optimizer': 'RMSprop'}
-56523900.000000 (2665037.687018) with: {'batch_size': 40, 'epochs': 10, 'optimizer': 'Adagrad'}
-56658.258203 (24003.537579) with: {'batch_size': 40, 'epochs': 50, 'optimizer': 'adam'}
-64086.296094 (12042.358310) with: {'batch_size': 40, 'epochs': 50, 'optimizer': 'RMSprop'}
-9372795.800000 (5108249.641949) with: {'batch_size': 40, 'epochs': 50, 'optimizer': 'Adagrad'}
-30622.471875 (6322.287248) with: {'batch_size': 40, 'epochs': 100, 'optimizer': 'adam'}
-36232.569531 (13259.656484) with: {'batch_size': 40, 'epochs': 100, 'optimizer': 'RMSprop'}
-1600181.925000 (146014.239422) with: {'batch_size': 40, 'epochs': 100, 'optimizer': 'Adagrad'}
-1390699.350000 (145640.273592) with: {'batch_size': 60, 'epochs': 10, 'optimizer': 'adam'}
-1082542.925000 (144731.452078) with: {'batch_size': 60, 'epochs': 10, 'optimizer': 'RMSprop'}
-62656396.800000 (559420.032519) with: {'batch_size': 60, 'epochs': 10, 'optimizer': 'Adagrad'}
-69710.080469 (40863.769851) with: {'batch_size': 60, 'epochs': 50, 'optimizer': 'adam'}
-71970.824219 (24058.956433) with: {'batch_size': 60, 'epochs': 50, 'optimizer': 'RMSprop'}
-16491987.400000 (3500092.027003) with: {'batch_size': 60, 'epochs': 50, 'optimizer': 'Adagrad'}
-46966.215625 (15952.838801) with: {'batch_size': 60, 'epochs': 100, 'optimizer': 'adam'}
-45104.332812 (10972.408712) with: {'batch_size': 60, 'epochs': 100, 'optimizer': 'RMSprop'}
```

-2788073.200000 (698682.820182) with: {'batch_size': 60, 'epochs': 100, 'optimizer': 'Adagrad'}
-1493044.875000 (155697.516601) with: {'batch_size': 80, 'epochs': 10, 'optimizer': 'adam'}
-1351079.800000 (130707.791587) with: {'batch_size': 80, 'epochs': 10, 'optimizer': 'RMSprop'}
-65509906.400000 (3248526.947553) with: {'batch_size': 80, 'epochs': 10, 'optimizer': 'Adagrad'}
-263853.200000 (123436.623595) with: {'batch_size': 80, 'epochs': 50, 'optimizer': 'adam'}
-92486.471875 (25669.353331) with: {'batch_size': 80, 'epochs': 50, 'optimizer': 'RMSprop'}
-25053901.200000 (2766136.455614) with: {'batch_size': 80, 'epochs': 50, 'optimizer': 'Adagrad'}
-41316.805469 (6963.559710) with: {'batch_size': 80, 'epochs': 100, 'optimizer': 'adam'}
-47747.921094 (15393.723483) with: {'batch_size': 80, 'epochs': 100, 'optimizer': 'RMSprop'}
-6449660.600000 (3418975.118244) with: {'batch_size': 80, 'epochs': 100, 'optimizer': 'Adagrad'}
-1476760.825000 (167679.598081) with: {'batch_size': 100, 'epochs': 10, 'optimizer': 'adam'}
-1404041.825000 (201396.916914) with: {'batch_size': 100, 'epochs': 10, 'optimizer': 'RMSprop'}
-72146363.200000 (2264547.064452) with: {'batch_size': 100, 'epochs': 10, 'optimizer': 'Adagrad'}
-352332.837500 (104710.011595) with: {'batch_size': 100, 'epochs': 50, 'optimizer': 'adam'}
-90365.727344 (25890.780854) with: {'batch_size': 100, 'epochs': 50, 'optimizer': 'RMSprop'}
-32726843.600000 (8646951.726295) with: {'batch_size': 100, 'epochs': 50, 'optimizer': 'Adagrad'}
-42565.274219 (14731.363104) with: {'batch_size': 100, 'epochs': 100, 'optimizer': 'adam'}
-65972.997656 (29357.657998) with: {'batch_size': 100, 'epochs': 100, 'optimizer': 'RMSprop'}
-11417867.800000 (2452926.970178) with: {'batch_size': 100, 'epochs': 100, 'optimizer': 'Adagrad'}

In [10]:

```
def custom_model( momentum=0, dropout_rate=0.0, learn_rate=0.01, epochs
= 10, verbose=0):
    model = Sequential()
    model.add(Dense(128, input_dim=X.shape[1], activation='relu'))
    model.add(Dense(64, activation='relu'))
    model.add(Dense(1))
    adam = Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None, decay=
0.0, amsgrad=False)
    model.compile(loss='mean_squared_error', optimizer=adam, metrics=['m
se'])
    return model

np.random.seed(5)

model = KerasRegressor(build_fn=custom_model, verbose=0)

# Hyperparameter tuning
learn_rate = [0.0001, 0.001, 0.01]
dropout_rate = [0.0, 0.2, 0.3]
batch_size = [10, 50, 100]
epochs = [10, 50, 100]

param_grid = dict(batch_size=batch_size, epochs=epochs, learn_rate=learn
_rate, dropout_rate=dropout_rate)

grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1)
grid_result = grid.fit(X, Y)
```

I then created a model with two dense layers and used the Adam optimizer to perform the remaining hyperparameter tuning. There were the outputs that were obtained

In [11]:
```
print("Best mse is %f with params --> %s" % (grid_result.best_score_, gr
id_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
std_dev = grid_result.cv_results_['std_test_score']
tuned_params = grid_result.cv_results_['params' ]
for mean, stdev, param in zip(means, std_dev, tuned_params):
    print("%f, %f ----> %r" % (mean, stdev, param))
```

```
Best mse is -24887.330078 with params --> {'batch_size': 10, 'dropout_r
ate': 0.2, 'epochs': 100, 'learn_rate': 0.01}
-83463.160156, 25271.865298 ----> {'batch_size': 10, 'dropout_rate': 0.
0, 'epochs': 10, 'learn_rate': 0.0001}
-88775.615625, 23038.250922 ----> {'batch_size': 10, 'dropout_rate': 0.
0, 'epochs': 10, 'learn_rate': 0.001}
-90300.914844, 32942.413488 ----> {'batch_size': 10, 'dropout_rate': 0.
0, 'epochs': 10, 'learn_rate': 0.01}
-36012.864453, 11941.555961 ----> {'batch_size': 10, 'dropout_rate': 0.
0, 'epochs': 50, 'learn_rate': 0.0001}
-31121.520313, 7992.348638 ----> {'batch_size': 10, 'dropout_rate': 0.
0, 'epochs': 50, 'learn_rate': 0.001}
-28983.807812, 5356.626577 ----> {'batch_size': 10, 'dropout_rate': 0.
0, 'epochs': 50, 'learn_rate': 0.01}
-35069.562109, 8180.334911 ----> {'batch_size': 10, 'dropout_rate': 0.
0, 'epochs': 100, 'learn_rate': 0.0001}
-33771.587500, 7284.982974 ----> {'batch_size': 10, 'dropout_rate': 0.
0, 'epochs': 100, 'learn_rate': 0.001}
-32479.938281, 8067.058798 ----> {'batch_size': 10, 'dropout_rate': 0.
0, 'epochs': 100, 'learn_rate': 0.01}
-65907.715625, 24826.923191 ----> {'batch_size': 10, 'dropout_rate': 0.
2, 'epochs': 10, 'learn_rate': 0.0001}
-77717.960156, 34020.840710 ----> {'batch_size': 10, 'dropout_rate': 0.
2, 'epochs': 10, 'learn_rate': 0.001}
-85224.619531, 29205.053937 ----> {'batch_size': 10, 'dropout_rate': 0.
2, 'epochs': 10, 'learn_rate': 0.01}
-33830.892578, 3986.515899 ----> {'batch_size': 10, 'dropout_rate': 0.
2, 'epochs': 50, 'learn_rate': 0.0001}
-31440.497656, 7374.794438 ----> {'batch_size': 10, 'dropout_rate': 0.
2, 'epochs': 50, 'learn_rate': 0.001}
-27606.241406, 4662.202180 ----> {'batch_size': 10, 'dropout_rate': 0.
2, 'epochs': 50, 'learn_rate': 0.01}
-29065.995703, 5938.747315 ----> {'batch_size': 10, 'dropout_rate': 0.
2, 'epochs': 100, 'learn_rate': 0.0001}
-26874.994922, 4138.167862 ----> {'batch_size': 10, 'dropout_rate': 0.
2, 'epochs': 100, 'learn_rate': 0.001}
-24887.330078, 3946.426631 ----> {'batch_size': 10, 'dropout_rate': 0.
2, 'epochs': 100, 'learn_rate': 0.01}
-70273.667188, 37229.902720 ----> {'batch_size': 10, 'dropout_rate': 0.
3, 'epochs': 10, 'learn_rate': 0.0001}
-107712.754687, 46457.342344 ----> {'batch_size': 10, 'dropout_rate':
0.3, 'epochs': 10, 'learn_rate': 0.001}
-79865.365625, 20842.438363 ----> {'batch_size': 10, 'dropout_rate': 0.
3, 'epochs': 10, 'learn_rate': 0.01}
-29312.818750, 6711.976675 ----> {'batch_size': 10, 'dropout_rate': 0.
3, 'epochs': 50, 'learn_rate': 0.0001}
-28767.895313, 2799.946145 ----> {'batch_size': 10, 'dropout_rate': 0.
3, 'epochs': 50, 'learn_rate': 0.001}
-33539.787500, 7869.685157 ----> {'batch_size': 10, 'dropout_rate': 0.
3, 'epochs': 50, 'learn_rate': 0.01}
-29265.059375, 4962.019223 ----> {'batch_size': 10, 'dropout_rate': 0.
3, 'epochs': 100, 'learn_rate': 0.0001}
-38803.658594, 26620.278088 ----> {'batch_size': 10, 'dropout_rate': 0.
3, 'epochs': 100, 'learn_rate': 0.001}
-26067.693750, 2738.324925 ----> {'batch_size': 10, 'dropout_rate': 0.
3, 'epochs': 100, 'learn_rate': 0.01}
-1391231.450000, 131769.763490 ----> {'batch_size': 50, 'dropout_rate':
```

```
0.0, 'epochs': 10, 'learn_rate': 0.0001}
-1394257.100000, 148163.606630 ----> {'batch_size': 50, 'dropout_rate':
0.0, 'epochs': 10, 'learn_rate': 0.001}
-1334672.825000, 121353.652625 ----> {'batch_size': 50, 'dropout_rate':
0.0, 'epochs': 10, 'learn_rate': 0.01}
-77671.030078, 72929.877871 ----> {'batch_size': 50, 'dropout_rate': 0.
0, 'epochs': 50, 'learn_rate': 0.0001}
-54267.003906, 26992.621148 ----> {'batch_size': 50, 'dropout_rate': 0.
0, 'epochs': 50, 'learn_rate': 0.001}
-53134.264062, 25623.205015 ----> {'batch_size': 50, 'dropout_rate': 0.
0, 'epochs': 50, 'learn_rate': 0.01}
-31013.926172, 3174.209647 ----> {'batch_size': 50, 'dropout_rate': 0.
0, 'epochs': 100, 'learn_rate': 0.0001}
-48720.391016, 16649.325040 ----> {'batch_size': 50, 'dropout_rate': 0.
0, 'epochs': 100, 'learn_rate': 0.001}
-31847.444141, 13980.987319 ----> {'batch_size': 50, 'dropout_rate': 0.
0, 'epochs': 100, 'learn_rate': 0.01}
-1345242.825000, 160264.462418 ----> {'batch_size': 50, 'dropout_rate':
0.2, 'epochs': 10, 'learn_rate': 0.0001}
-1343731.025000, 142865.047342 ----> {'batch_size': 50, 'dropout_rate':
0.2, 'epochs': 10, 'learn_rate': 0.001}
-1354697.775000, 124586.394037 ----> {'batch_size': 50, 'dropout_rate':
0.2, 'epochs': 10, 'learn_rate': 0.01}
-57313.091016, 18280.070121 ----> {'batch_size': 50, 'dropout_rate': 0.
2, 'epochs': 50, 'learn_rate': 0.0001}
-51404.878125, 26886.678269 ----> {'batch_size': 50, 'dropout_rate': 0.
2, 'epochs': 50, 'learn_rate': 0.001}
-53362.170312, 20306.481582 ----> {'batch_size': 50, 'dropout_rate': 0.
2, 'epochs': 50, 'learn_rate': 0.01}
-31464.076953, 6984.338670 ----> {'batch_size': 50, 'dropout_rate': 0.
2, 'epochs': 100, 'learn_rate': 0.0001}
-34847.044141, 15228.993071 ----> {'batch_size': 50, 'dropout_rate': 0.
2, 'epochs': 100, 'learn_rate': 0.001}
-32029.678125, 5482.817316 ----> {'batch_size': 50, 'dropout_rate': 0.
2, 'epochs': 100, 'learn_rate': 0.01}
-1379998.125000, 172412.173111 ----> {'batch_size': 50, 'dropout_rate':
0.3, 'epochs': 10, 'learn_rate': 0.0001}
-1378416.150000, 94280.468042 ----> {'batch_size': 50, 'dropout_rate':
0.3, 'epochs': 10, 'learn_rate': 0.001}
-1372643.625000, 120326.731641 ----> {'batch_size': 50, 'dropout_rate':
0.3, 'epochs': 10, 'learn_rate': 0.01}
-56712.373437, 11561.815221 ----> {'batch_size': 50, 'dropout_rate': 0.
3, 'epochs': 50, 'learn_rate': 0.0001}
-64784.871875, 12282.235965 ----> {'batch_size': 50, 'dropout_rate': 0.
3, 'epochs': 50, 'learn_rate': 0.001}
-69941.793750, 38007.654682 ----> {'batch_size': 50, 'dropout_rate': 0.
3, 'epochs': 50, 'learn_rate': 0.01}
-37416.461328, 13574.281323 ----> {'batch_size': 50, 'dropout_rate': 0.
3, 'epochs': 100, 'learn_rate': 0.0001}
-45251.885156, 16110.994895 ----> {'batch_size': 50, 'dropout_rate': 0.
3, 'epochs': 100, 'learn_rate': 0.001}
-29959.032031, 6280.830413 ----> {'batch_size': 50, 'dropout_rate': 0.
3, 'epochs': 100, 'learn_rate': 0.01}
-1494333.325000, 171158.091437 ----> {'batch_size': 100, 'dropout_rat
e': 0.0, 'epochs': 10, 'learn_rate': 0.0001}
-1492897.475000, 136893.803305 ----> {'batch_size': 100, 'dropout_rat
e': 0.0, 'epochs': 10, 'learn_rate': 0.001}
```

```
-1481787.225000, 181406.314004 ----> {'batch_size': 100, 'dropout_rat
e': 0.0, 'epochs': 10, 'learn_rate': 0.01}
-362559.275000, 144505.187410 ----> {'batch_size': 100, 'dropout_rate':
0.0, 'epochs': 50, 'learn_rate': 0.0001}
-493743.925000, 175628.419863 ----> {'batch_size': 100, 'dropout_rate':
0.0, 'epochs': 50, 'learn_rate': 0.001}
-425601.634375, 138994.073447 ----> {'batch_size': 100, 'dropout_rate':
0.0, 'epochs': 50, 'learn_rate': 0.01}
-56394.031250, 29822.650123 ----> {'batch_size': 100, 'dropout_rate':
0.0, 'epochs': 100, 'learn_rate': 0.0001}
-62118.402734, 33738.726838 ----> {'batch_size': 100, 'dropout_rate':
0.0, 'epochs': 100, 'learn_rate': 0.001}
-45354.618750, 12476.379207 ----> {'batch_size': 100, 'dropout_rate':
0.0, 'epochs': 100, 'learn_rate': 0.01}
-1520627.550000, 153316.697647 ----> {'batch_size': 100, 'dropout_rat
e': 0.2, 'epochs': 10, 'learn_rate': 0.0001}
-1525229.250000, 192898.677445 ----> {'batch_size': 100, 'dropout_rat
e': 0.2, 'epochs': 10, 'learn_rate': 0.001}
-1576432.075000, 140458.977827 ----> {'batch_size': 100, 'dropout_rat
e': 0.2, 'epochs': 10, 'learn_rate': 0.01}
-438015.534375, 169262.908999 ----> {'batch_size': 100, 'dropout_rate':
0.2, 'epochs': 50, 'learn_rate': 0.0001}
-526000.006250, 172199.747528 ----> {'batch_size': 100, 'dropout_rate':
0.2, 'epochs': 50, 'learn_rate': 0.001}
-302433.350000, 203837.546876 ----> {'batch_size': 100, 'dropout_rate':
0.2, 'epochs': 50, 'learn_rate': 0.01}
-65191.571875, 16258.684909 ----> {'batch_size': 100, 'dropout_rate':
0.2, 'epochs': 100, 'learn_rate': 0.0001}
-65105.631250, 21753.605495 ----> {'batch_size': 100, 'dropout_rate':
0.2, 'epochs': 100, 'learn_rate': 0.001}
-42338.783203, 14474.060719 ----> {'batch_size': 100, 'dropout_rate':
0.2, 'epochs': 100, 'learn_rate': 0.01}
-1580144.925000, 167657.703649 ----> {'batch_size': 100, 'dropout_rat
e': 0.3, 'epochs': 10, 'learn_rate': 0.0001}
-1572937.625000, 189683.774624 ----> {'batch_size': 100, 'dropout_rat
e': 0.3, 'epochs': 10, 'learn_rate': 0.001}
-1567378.325000, 161370.750398 ----> {'batch_size': 100, 'dropout_rat
e': 0.3, 'epochs': 10, 'learn_rate': 0.01}
-461903.221875, 243098.024866 ----> {'batch_size': 100, 'dropout_rate':
0.3, 'epochs': 50, 'learn_rate': 0.0001}
-329203.254688, 191314.508843 ----> {'batch_size': 100, 'dropout_rate':
0.3, 'epochs': 50, 'learn_rate': 0.001}
-514557.693750, 125854.978822 ----> {'batch_size': 100, 'dropout_rate':
0.3, 'epochs': 50, 'learn_rate': 0.01}
-68567.199219, 37323.214715 ----> {'batch_size': 100, 'dropout_rate':
0.3, 'epochs': 100, 'learn_rate': 0.0001}
-46243.672266, 26302.250809 ----> {'batch_size': 100, 'dropout_rate':
0.3, 'epochs': 100, 'learn_rate': 0.001}
-79122.269922, 52110.550128 ----> {'batch_size': 100, 'dropout_rate':
0.3, 'epochs': 100, 'learn_rate': 0.01}
```

From the above values, we notice that the most optimal set of attributes were found to be.

'batch_size': 10, 'dropout_rate': 0.2, 'epochs': 100, 'learn_rate': 0.01

Task 2 - Compare the trained neural networkwith multivariable regression

In [16]:
```
X2 = sm.add_constant(X)
est = sm.OLS(Y, X2)
est2 = est.fit()
```

In [17]:
```
print(est2.summary())
```

```
                            OLS Regression Results
========================================================================
=======
Dep. Variable:                      y   R-squared:
0.960
Model:                            OLS   Adj. R-squared:
0.960
Method:                 Least Squares   F-statistic:
9585.
Date:                Thu, 05 Nov 2020   Prob (F-statistic):
0.00
Time:                        01:55:01   Log-Likelihood:
-14158.
No. Observations:                2000   AIC:                           2.
833e+04
Df Residuals:                    1994   BIC:                           2.
836e+04
Df Model:                           5
Covariance Type:            nonrobust
========================================================================
=======
                 coef    std err          t      P>|t|      [0.025
0.975]
------------------------------------------------------------------------
-------
const      -2567.9162    142.217    -18.056      0.000   -2846.826   -2
289.007
x1            55.0369      0.259    212.402      0.000      54.529
55.545
x2             2.2014      0.267      8.252      0.000       1.678
2.725
x3             5.6969      0.266     21.387      0.000       5.175
6.219
x4             6.9531      0.251     27.745      0.000       6.462
7.445
x5             9.1432      0.263     34.767      0.000       8.627
9.659
========================================================================
=======
Omnibus:                     1395.967   Durbin-Watson:
1.974
Prob(Omnibus):                  0.000   Jarque-Bera (JB):             29
299.309
Skew:                           3.018   Prob(JB):
0.00
Kurtosis:                      20.753   Cond. No.
1.22e+04
========================================================================
=======

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is
correctly specified.
[2] The condition number is large, 1.22e+04. This might indicate that t
here are
strong multicollinearity or other numerical problems.
```

In [18]:
```
reg2 = LinearRegression()
reg2.fit(X, Y)

print("The linear model is: Y = {:.5} + {:.5}*X1 + {:.5}*X2 + {:.5}*X3 +
{:.5}*X4 + {:.5}*X5".format(reg2.intercept_, reg2.coef_[0], reg2.coef_[1
], reg2.coef_[2], reg2.coef_[3], reg2.coef_[4]))
print("Y = a0 + a1X1 + a3X3 + a4X4 + a5X5")
```

```
The linear model is: Y = -2567.9 + 55.037*X1 + 2.2014*X2 + 5.6969*X3 +
6.9531*X4 + 9.1432*X5
Y = a0 + a1X1 + a3X3 + a4X4 + a5X5
```

We now calculate the sum of squared errors (SSE) for each of the models and determine which is the better model

In [19]:
```
LR_sse = 0
for v in Y - reg2.predict(X):
    LR_sse += v**2
```

In [20]:
```
NN_sse = 0
for v in Y - grid_result.predict(X):
    NN_sse += v**2
```

In [21]:
```
print("SSE for Multivariate regression: ", LR_sse)
print("SSE for estimation with Neural Moedl: ", NN_sse)
```

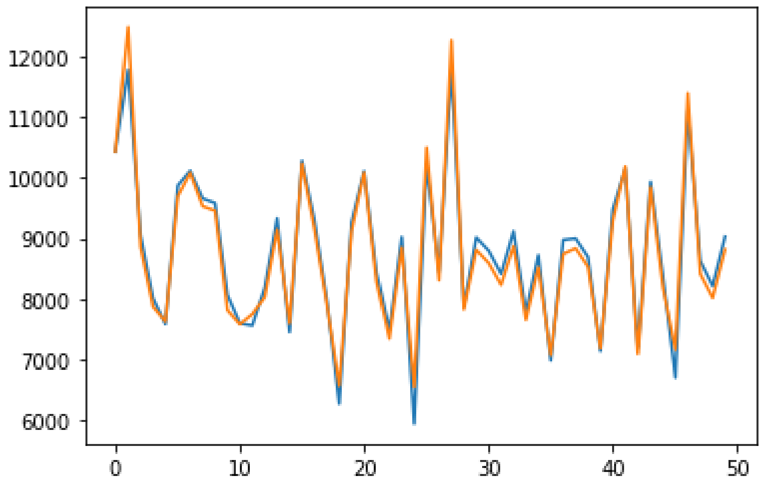
```
SSE for Multivariate regression:  164973673.90797538
SSE for estimation with Neural Moedl:  44258448.18429801
```

It can be seen that the SSE value for the custom neural model created with hyperparameter tuning seems to fare better in comparison to the Multivariable linear regression.

Below are two sample predictions made on untrained test data by both the models. To plain sight, the difference is minimal but on further analysis with hyper parammeter tuning, we see a much bigger difference in performance between the two models.

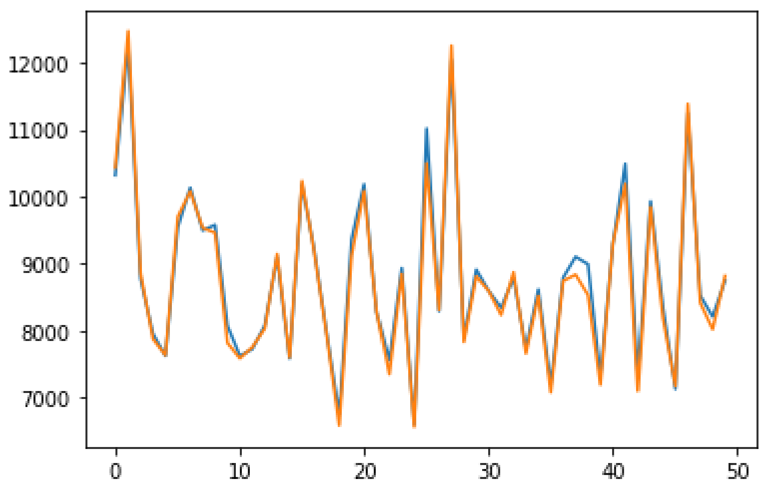
In [22]:
```
Y_test_pred = reg2.predict(X_test)
plt.plot(Y_test_pred[:50])
plt.plot(Y_test[:50])
```

Out[22]: [<matplotlib.lines.Line2D at 0x7faf265dbdd8>]



In [23]:
```
Y_test_pred_NN = grid_result.predict(X_test)
plt.plot(Y_test_pred_NN[:50])
plt.plot(Y_test[:50])
```

Out[23]: [<matplotlib.lines.Line2D at 0x7faf2877f198>]



Conclusions

We notice that hyperparameter tuning is important and upon proper analysis choice of the parameters, a neural model can perform better than the previously run Multivariable regression model.

Refs: https://machinelearningmastery.com/tutorial-first-neural-network-python-keras/ (https://machinelearningmastery.com/tutorial-first-neural-network-python-keras/)

https://machinelearningmastery.com/regression-tutorial-keras-deep-learning-library-python/ (https://machinelearningmastery.com/regression-tutorial-keras-deep-learning-library-python/)

https://www.kaggle.com/willkoehrsen/intro-to-model-tuning-grid-and-random-search (https://www.kaggle.com/willkoehrsen/intro-to-model-tuning-grid-and-random-search)