ATM Cash Reloading Forecasting

Background

In INDIA, there are a total of 66 banks including 27 PSBs [19 Nationalized banks + 6 State bank group (SBI + 5 associates) + 1 IDBI bank (Other Public Sector-Indian Bank) = 26 PSBs + 1 Bhartiya Mahila Bank, 39 Private Limited Banks]

PAN INDIA we have 213004 ATMs available, performing average of 80,55,22,146 monthly transactions, of 27,59,761 Million rupees.

INDIA has one of highest ATM utilization rate in world. Any Downtime in ATM means inconvenience for customers.

Each bank his own ATM replenishment network to maintain ATMs and adequate Balance for hassles services. Most of the Banks outsource the process of re-filling the ATM process.

It is always possible that

> ATM is overloaded:
>
> - Money that is not required for the ATM for the work day/week
> - Dumping the excessive money that could be used in Banks daily cash rotation
> - Wasting a day's interest of excessive loaded money
>
> Low Cash:
>
> - Not maintaining the adequate money required to run the ATM till next scheduled replenishment
> - Causing the frequent load - increasing the man power and fuel cost for round trip from Bank to ATM

Goal

The goal of the MVP is to build an adaptive algorithm which predicts the adequate cash required for the ATM to be online till next scheduled re-fill. Provide 24/7 up-time, ZERO downtime Decrease the overloaded cash in ATM Increase the Cash Rotation in Bank More cash availability to Bank means - Bank can use the money dumped in ATM to loan customers and make an interest income

Building a Predictive Analysis

- Using Python and Existing data of ATM relod we are building a predictive analytic model to predict the cash required for the ATM run for the day till next schedulded ATM replenishment

Data

- We used 4 years masked data of a Private Bank ATM located at XYZ Road

Import the libraries

In [15]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

Step 1 : Data Processing

Import the dataset - Data of ATMs

In [97]:
```
dataset = pd.read_csv('C:/Users/37291/Downloads/atm_data_m2.csv')
dataset[:5]
```

Out[97]:

	Unnamed: 0	atm_name	weekday	festival_religion	working_day	holiday_sequenc
0	11	Mount Road ATM	MONDAY	NH	W	WWW
1	16	Mount Road ATM	TUESDAY	NH	W	WWW
2	21	Mount Road ATM	WEDNESDAY	NH	W	WWW
3	26	Mount Road ATM	THURSDAY	NH	W	WWW
4	31	Mount Road ATM	FRIDAY	NH	W	WWW

Separating Target Variable

In [98]:
```
X = dataset.iloc[:, 1:10].values
y = dataset.iloc[:, 10].values
```

Encoding Categorical Data into Number for Processing

In [99]:
```
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
labelencoder_X_0 = LabelEncoder()
X[:,0] = labelencoder_X_0.fit_transform(X[:, 0])
labelencoder_X_1 = LabelEncoder()
X[:,1] = labelencoder_X_1.fit_transform(X[:, 1])
labelencoder_X_2 = LabelEncoder()
X[:,2] = labelencoder_X_2.fit_transform(X[:, 2])
labelencoder_X_3 = LabelEncoder()
X[:,3] = labelencoder_X_3.fit_transform(X[:, 3])
labelencoder_X_4 = LabelEncoder()
X[:,4] = labelencoder_X_4.fit_transform(X[:, 4])
labelencoder_X_5 = LabelEncoder()
X[:,5] = labelencoder_X_5.fit_transform(X[:, 5])
```

Encode variable with one hotencoder

In [100]:
```
onehotencoder = OneHotEncoder(categorical_features = [1,2,3,4,5])
X = onehotencoder.fit_transform(X).toarray()
X
```

Out[100]:
```
array([[  0.00000000e+00,   1.00000000e+00,   0.00000000e+00, ...,
          1.00000000e+00,   2.01100000e+03,   6.48600000e+05],
       [  0.00000000e+00,   0.00000000e+00,   0.00000000e+00, ...,
          1.00000000e+00,   2.01100000e+03,   6.48600000e+05],
       [  0.00000000e+00,   0.00000000e+00,   0.00000000e+00, ...,
          1.00000000e+00,   2.01100000e+03,   6.48600000e+05],
       ...,
       [  0.00000000e+00,   0.00000000e+00,   0.00000000e+00, ...,
          9.00000000e+00,   2.01700000e+03,   2.76058000e+05],
       [  0.00000000e+00,   0.00000000e+00,   0.00000000e+00, ...,
          9.00000000e+00,   2.01700000e+03,   2.76058000e+05],
       [  1.00000000e+00,   0.00000000e+00,   0.00000000e+00, ...,
          9.00000000e+00,   2.01700000e+03,   2.76058000e+05]])
```

Splitting the dataset into the Training set and Test set

In [101]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, ran
dom_state = 0)
```

Predictive Model 1 - Running Multi Linear Regression Alrogithm

Fitting Multiple Linear Regression to the Training set

In [33]:
```
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
regressor.fit(X_train, y_train)
```

Out[33]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

Predicting the Test set results

In [35]:
```
y_pred = regressor.predict(X_test)
```

Print the predicted vs actual results

In [40]:
```
for i in range(10):
    print("Y=%s, Predicted=%s" % (y_test[i], y_pred[i]))
```

```
Y=530100, Predicted=511507.169922
Y=930900, Predicted=752278.907227
Y=781900, Predicted=578784.726563
Y=350900, Predicted=463162.506836
Y=462500, Predicted=520832.549805
Y=60600, Predicted=336871.716797
Y=703300, Predicted=742320.213867
Y=311900, Predicted=590765.365234
Y=546100, Predicted=591644.558594
Y=305000, Predicted=208484.222656
```

Making the Confusion Matrix to check accuracy of Model

In [44]:
```
from sklearn import metrics
from sklearn import metrics
print(metrics.mean_absolute_error(y_test, y_pred))
print(metrics.mean_squared_error(y_test, y_pred))
print(np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
(np.sqrt(metrics.mean_squared_error(y_test, y_pred))/
np.mean(y_test))
```

```
142981.203967
33999103988.0
184388.459476
```

Out[44]:
```
0.37116111245019862
```

Using Model 1 we have got *37%* accuracy

Predictive Model 2 - Artificial Neural Networks (ANN)

Feature Scaling

In [102]:
```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

Importing the Keras libraries and packages (TensoFlow)

In [103]:
```
import keras
from keras.models import Sequential
from keras.layers import Dense
classifier = Sequential()
```

Adding the input layer and the first hidden layer

In [104]:
```
classifier.add(Dense(output_dim = 25, init = 'uniform', activation = 'relu', i
nput_dim = 33))
```

```
C:\Users\37291\AppData\Local\conda\conda\envs\mypython3\lib\site-packages\ipy
kernel_launcher.py:1: UserWarning: Update your `Dense` call to the Keras 2 AP
I: `Dense(activation="relu", input_dim=33, units=25, kernel_initializer="unif
orm")`
  """Entry point for launching an IPython kernel.
```

Adding the 2 hidden layer

In [105]:
```
classifier.add(Dense(output_dim = 20, init = 'uniform', activation = 'relu'))
```

```
C:\Users\37291\AppData\Local\conda\conda\envs\mypython3\lib\site-packages\ipy
kernel_launcher.py:1: UserWarning: Update your `Dense` call to the Keras 2 AP
I: `Dense(activation="relu", units=20, kernel_initializer="uniform")`
  """Entry point for launching an IPython kernel.
```

Adding the 3 hidden layer

In [106]:
```
classifier.add(Dense(output_dim = 15, init = 'uniform', activation = 'relu'))
```

```
C:\Users\37291\AppData\Local\conda\conda\envs\mypython3\lib\site-packages\ipy
kernel_launcher.py:1: UserWarning: Update your `Dense` call to the Keras 2 AP
I: `Dense(activation="relu", units=15, kernel_initializer="uniform")`
  """Entry point for launching an IPython kernel.
```

Adding the 4 hidden layer

In [107]:
```
classifier.add(Dense(output_dim = 8, init = 'uniform', activation = 'relu'))
```

```
C:\Users\37291\AppData\Local\conda\conda\envs\mypython3\lib\site-packages\ipy
kernel_launcher.py:1: UserWarning: Update your `Dense` call to the Keras 2 AP
I: `Dense(activation="relu", units=8, kernel_initializer="uniform")`
  """Entry point for launching an IPython kernel.
```

Adding the output layer

In [108]:
```
classifier.add(Dense(output_dim = 1, init = 'uniform', activation = 'linear'))
```

```
C:\Users\37291\AppData\Local\conda\conda\envs\mypython3\lib\site-packages\ipy
kernel_launcher.py:1: UserWarning: Update your `Dense` call to the Keras 2 AP
I: `Dense(activation="linear", units=1, kernel_initializer="uniform")`
  """Entry point for launching an IPython kernel.
```

Adding the output layer

In [109]:
```
classifier.add(Dense(output_dim = 1, init = 'uniform', activation = 'linear'))
```

```
C:\Users\37291\AppData\Local\conda\conda\envs\mypython3\lib\site-packages\ipy
kernel_launcher.py:1: UserWarning: Update your `Dense` call to the Keras 2 AP
I: `Dense(activation="linear", units=1, kernel_initializer="uniform")`
  """Entry point for launching an IPython kernel.
```

Compiling the ANN

In [110]:
```
classifier.compile(optimizer = 'adam', loss = 'mse')
```

Fitting the ANN to the Training set

In [111]:

```
classifier.fit(X_train, y_train, batch_size = 10, nb_epoch = 100)
```

```
C:\Users\37291\AppData\Local\conda\conda\envs\mypython3\lib\site-packages\ker
as\models.py:939: UserWarning: The `nb_epoch` argument in `fit` has been rena
med `epochs`.
  warnings.warn('The `nb_epoch` argument in `fit` '
```

```
Epoch 1/100
1795/1795 [==============================] - 1s 772us/step - loss: 3368130479
90.5515
Epoch 2/100
1795/1795 [==============================] - 0s 130us/step - loss: 3310181727
29.7604
Epoch 3/100
1795/1795 [==============================] - 0s 136us/step - loss: 2210059968
76.3008
Epoch 4/100
1795/1795 [==============================] - 0s 128us/step - loss: 6316270424
8.5125
Epoch 5/100
1795/1795 [==============================] - 0s 135us/step - loss: 4762607801
1.1866
Epoch 6/100
1795/1795 [==============================] - 0s 135us/step - loss: 4430274668
6.7521
Epoch 7/100
1795/1795 [==============================] - 0s 129us/step - loss: 4217213936
5.9721
Epoch 8/100
1795/1795 [==============================] - 0s 139us/step - loss: 4057580210
8.4345
Epoch 9/100
1795/1795 [==============================] - 0s 128us/step - loss: 3920076922
3.6657
Epoch 10/100
1795/1795 [==============================] - 0s 148us/step - loss: 3822672104
1.8273
Epoch 11/100
1795/1795 [==============================] - 0s 121us/step - loss: 3731011780
8.1337
Epoch 12/100
1795/1795 [==============================] - 0s 133us/step - loss: 3670091308
7.8217
Epoch 13/100
1795/1795 [==============================] - 0s 137us/step - loss: 3622459136
9.9833
Epoch 14/100
1795/1795 [==============================] - 0s 130us/step - loss: 3580094762
6.4290
Epoch 15/100
1795/1795 [==============================] - 0s 126us/step - loss: 3543973257
0.5627
Epoch 16/100
1795/1795 [==============================] - 0s 140us/step - loss: 3518401644
3.8997
Epoch 17/100
1795/1795 [==============================] - 0s 136us/step - loss: 3496424888
4.0557
Epoch 18/100
1795/1795 [==============================] - 0s 133us/step - loss: 3478717219
2.2674
Epoch 19/100
1795/1795 [==============================] - 0s 122us/step - loss: 3458983257
2.7911
```

```
Epoch 20/100
1795/1795 [==============================] - 0s 150us/step - loss: 3450897750
3.5543
Epoch 21/100
1795/1795 [==============================] - 0s 145us/step - loss: 3431020502
4.9805
Epoch 22/100
1795/1795 [==============================] - 0s 126us/step - loss: 3423321630
3.7772
Epoch 23/100
1795/1795 [==============================] - 0s 129us/step - loss: 3417044668
5.4150
Epoch 24/100
1795/1795 [==============================] - 0s 152us/step - loss: 3401071551
5.3649
Epoch 25/100
1795/1795 [==============================] - 0s 152us/step - loss: 3395243486
6.2730
Epoch 26/100
1795/1795 [==============================] - 0s 125us/step - loss: 3396291422
5.5599
Epoch 27/100
1795/1795 [==============================] - 0s 140us/step - loss: 3383499533
2.6351
Epoch 28/100
1795/1795 [==============================] - 0s 134us/step - loss: 3374231777
9.0752
Epoch 29/100
1795/1795 [==============================] - 0s 130us/step - loss: 3368724374
7.4763
Epoch 30/100
1795/1795 [==============================] - 0s 125us/step - loss: 3368120551
0.4178
Epoch 31/100
1795/1795 [==============================] - 0s 128us/step - loss: 3356531989
9.6323
Epoch 32/100
1795/1795 [==============================] - 0s 129us/step - loss: 3357633640
8.2451
Epoch 33/100
1795/1795 [==============================] - 0s 131us/step - loss: 3349125715
4.3175
Epoch 34/100
1795/1795 [==============================] - 0s 140us/step - loss: 3342756800
1.0696
Epoch 35/100
1795/1795 [==============================] - 0s 152us/step - loss: 3343003360
4.8134
Epoch 36/100
1795/1795 [==============================] - 0s 145us/step - loss: 3342486802
9.6825
Epoch 37/100
1795/1795 [==============================] - 0s 136us/step - loss: 3332209296
6.1504
Epoch 38/100
1795/1795 [==============================] - 0s 153us/step - loss: 3332796776
5.3928
```

```
Epoch 39/100
1795/1795 [==============================] - 0s 130us/step - loss: 3328916814
5.8273
Epoch 40/100
1795/1795 [==============================] - 0s 130us/step - loss: 3317864473
6.7131
Epoch 41/100
1795/1795 [==============================] - 0s 129us/step - loss: 3317885770
0.1894
Epoch 42/100
1795/1795 [==============================] - 0s 130us/step - loss: 3321218155
5.3426
Epoch 43/100
1795/1795 [==============================] - 0s 132us/step - loss: 3313998601
8.4067
Epoch 44/100
1795/1795 [==============================] - 0s 179us/step - loss: 3310465162
6.2507
Epoch 45/100
1795/1795 [==============================] - 0s 128us/step - loss: 3304749399
7.1031
Epoch 46/100
1795/1795 [==============================] - 0s 132us/step - loss: 3304466078
3.0641
Epoch 47/100
1795/1795 [==============================] - 0s 126us/step - loss: 3303931838
1.1031
Epoch 48/100
1795/1795 [==============================] - 0s 134us/step - loss: 3299410378
9.4596
Epoch 49/100
1795/1795 [==============================] - 0s 130us/step - loss: 3293622915
9.2201
Epoch 50/100
1795/1795 [==============================] - 0s 130us/step - loss: 3290695195
9.5320
Epoch 51/100
1795/1795 [==============================] - 0s 131us/step - loss: 3295481248
4.4568
Epoch 52/100
1795/1795 [==============================] - 0s 208us/step - loss: 3296452458
1.0808
Epoch 53/100
1795/1795 [==============================] - 0s 181us/step - loss: 3287212057
3.8607
Epoch 54/100
1795/1795 [==============================] - 0s 181us/step - loss: 3286314598
1.1476
Epoch 55/100
1795/1795 [==============================] - 0s 198us/step - loss: 3288227754
7.1421
Epoch 56/100
1795/1795 [==============================] - 0s 193us/step - loss: 3285164950
7.4763
Epoch 57/100
1795/1795 [==============================] - 0s 189us/step - loss: 3279890462
5.2033
```

```
Epoch 58/100
1795/1795 [==============================] - 0s 186us/step - loss: 3277578142
7.3426
Epoch 59/100
1795/1795 [==============================] - 0s 137us/step - loss: 3277559220
9.8273
Epoch 60/100
1795/1795 [==============================] - 0s 154us/step - loss: 3270439832
7.4429 0s - loss: 28850960429
Epoch 61/100
1795/1795 [==============================] - 0s 138us/step - loss: 3278493921
0.5181
Epoch 62/100
1795/1795 [==============================] - 0s 149us/step - loss: 3262182833
5.5989
Epoch 63/100
1795/1795 [==============================] - 0s 134us/step - loss: 3282268661
8.7409
Epoch 64/100
1795/1795 [==============================] - 0s 151us/step - loss: 3271349209
7.7827
Epoch 65/100
1795/1795 [==============================] - 0s 130us/step - loss: 3267084373
5.7103
Epoch 66/100
1795/1795 [==============================] - 0s 170us/step - loss: 3261455119
2.6017
Epoch 67/100
1795/1795 [==============================] - 0s 192us/step - loss: 3257336904
4.5014
Epoch 68/100
1795/1795 [==============================] - 0s 171us/step - loss: 3256412920
7.2646
Epoch 69/100
1795/1795 [==============================] - 0s 138us/step - loss: 3255269820
6.8412
Epoch 70/100
1795/1795 [==============================] - 0s 135us/step - loss: 3261737632
0.1783
Epoch 71/100
1795/1795 [==============================] - 0s 135us/step - loss: 3248520687
1.7103
Epoch 72/100
1795/1795 [==============================] - 0s 144us/step - loss: 3253169568
7.3092
Epoch 73/100
1795/1795 [==============================] - 0s 151us/step - loss: 3247201213
8.2507
Epoch 74/100
1795/1795 [==============================] - 0s 155us/step - loss: 3249791685
9.5432
Epoch 75/100
1795/1795 [==============================] - 0s 130us/step - loss: 3249364698
7.7660
Epoch 76/100
1795/1795 [==============================] - 0s 152us/step - loss: 3238868042
4.4680
```

```
Epoch 77/100
1795/1795 [==============================] - 0s 146us/step - loss: 3240157225
3.5933
Epoch 78/100
1795/1795 [==============================] - 0s 149us/step - loss: 3238657120
8.3788
Epoch 79/100
1795/1795 [==============================] - 0s 142us/step - loss: 3246640231
8.2618
Epoch 80/100
1795/1795 [==============================] - 0s 151us/step - loss: 3235030964
1.2702
Epoch 81/100
1795/1795 [==============================] - 0s 135us/step - loss: 3238933301
1.9666
Epoch 82/100
1795/1795 [==============================] - 0s 158us/step - loss: 3235657003
4.9861
Epoch 83/100
1795/1795 [==============================] - 0s 141us/step - loss: 3234685320
9.1365
Epoch 84/100
1795/1795 [==============================] - 0s 154us/step - loss: 3235789694
5.0251
Epoch 85/100
1795/1795 [==============================] - 0s 133us/step - loss: 3229943493
6.6908
Epoch 86/100
1795/1795 [==============================] - 0s 128us/step - loss: 3225794980
7.2423
Epoch 87/100
1795/1795 [==============================] - 0s 123us/step - loss: 3236232416
1.9610
Epoch 88/100
1795/1795 [==============================] - 0s 155us/step - loss: 3223718636
1.7604
Epoch 89/100
1795/1795 [==============================] - 0s 112us/step - loss: 3219448965
2.0557
Epoch 90/100
1795/1795 [==============================] - 0s 142us/step - loss: 3226269282
9.7716
Epoch 91/100
1795/1795 [==============================] - 0s 121us/step - loss: 3222084003
8.6852
Epoch 92/100
1795/1795 [==============================] - 0s 121us/step - loss: 3218151347
5.5655
Epoch 93/100
1795/1795 [==============================] - 0s 121us/step - loss: 3216873179
0.6184
Epoch 94/100
1795/1795 [==============================] - 0s 128us/step - loss: 3215582291
0.0390
Epoch 95/100
1795/1795 [==============================] - 0s 152us/step - loss: 3219378620
6.8412
```

```
Epoch 96/100
1795/1795 [==============================] - 0s 147us/step - loss: 3219556202
8.2117
Epoch 97/100
1795/1795 [==============================] - 0s 124us/step - loss: 3210847999
2.8691
Epoch 98/100
1795/1795 [==============================] - 0s 118us/step - loss: 3209566579
3.7827
Epoch 99/100
1795/1795 [==============================] - 0s 150us/step - loss: 3210141651
5.0306
Epoch 100/100
1795/1795 [==============================] - 0s 120us/step - loss: 3210809112
1.0251
```

Out[111]:
```
<keras.callbacks.History at 0xf51deb8>
```

Part 3 - Making the predictions and evaluating the model

Predicting the Test set results

In [113]:
```
y_ANN_pred = classifier.predict(X_test)
```

Print the predicted vs actual results

In [114]:
```
for i in range(10):
    print("Y=%s, Predicted=%s" % (y_test[i], y_ANN_pred[i]))
```

```
Y=530100, Predicted=[ 486341.40625]
Y=930900, Predicted=[ 773025.125]
Y=781900, Predicted=[ 560214.3125]
Y=350900, Predicted=[ 437500.09375]
Y=462500, Predicted=[ 386823.78125]
Y=60600, Predicted=[ 344038.0625]
Y=703300, Predicted=[ 741862.]
Y=311900, Predicted=[ 560975.875]
Y=546100, Predicted=[ 546977.75]
Y=305000, Predicted=[ 298328.4375]
```

Making the Confusion Matrix to check accuracy of Model

In [115]:
```
from sklearn import metrics

print(metrics.mean_absolute_error(y_test, y_ANN_pred))
print(metrics.mean_squared_error(y_test, y_ANN_pred))
print(np.sqrt(metrics.mean_squared_error(y_test, y_ANN_pred)))
(np.sqrt(metrics.mean_squared_error(y_test, y_ANN_pred))/
np.mean(y_test))
```

```
139502.652996
32315714183.4
179765.720268
```

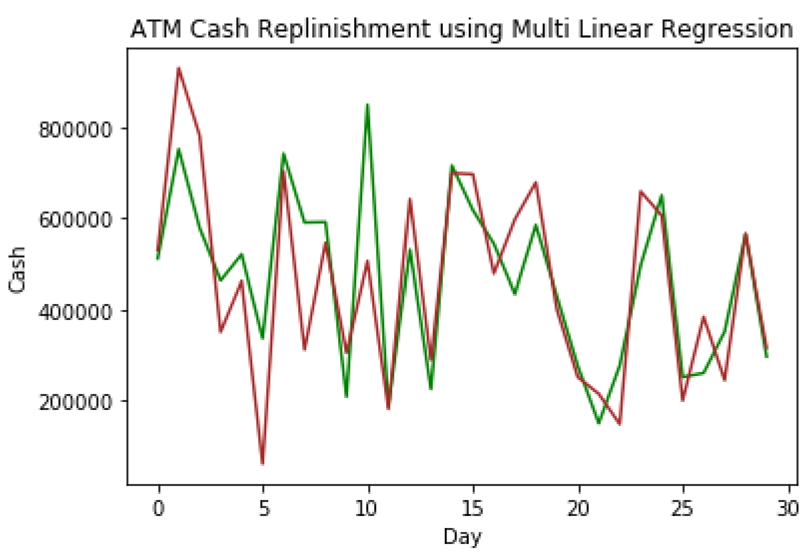
Out[115]:
```
0.36185586074401949
```

Using Model 2 we have got *36%* accuracy

Model 1 vs Model 2

Model 1 : Actual vs Predicted

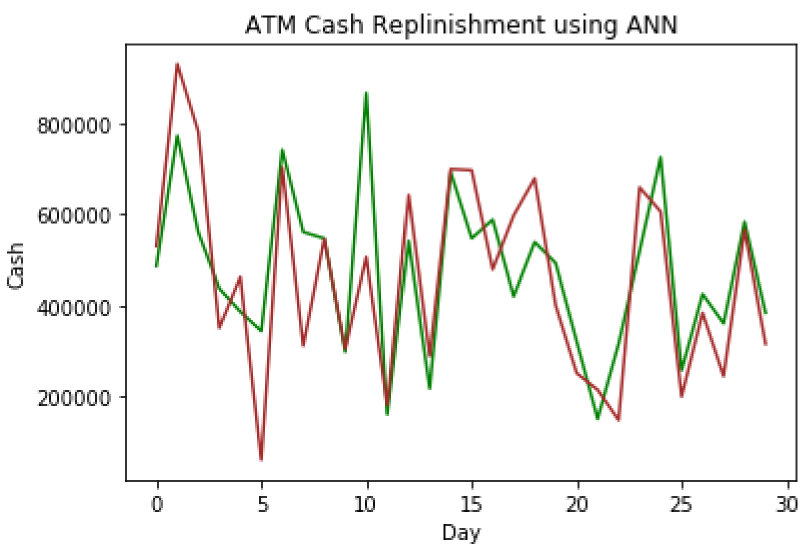
In [119]:
```
import matplotlib.pyplot as plt
plt.plot(y_pred[:30], color='green')
plt.plot(y_test[:30], color='brown')
plt.xlabel('Day')
plt.ylabel('Cash')
plt.title('ATM Cash Replinishment using Multi Linear Regression')
plt.show()
```



```
Brown Line - Actual Values  |  Green Line - Predicted Values
```

Model 2 : Actual vs Predicted

In [121]:
```
plt.plot(y_ANN_pred[:30], color='g')
plt.plot(y_test[:30], color='brown')
plt.xlabel('Day')
plt.ylabel('Cash')
plt.title('ATM Cash Replinishment using ANN')
plt.show()
```




```
Brown Line - Actual Values  |  Green Line - Predicted Values
```

Conclusion:

- Multi Linear Regression technique gave highest accuracy of 37% compare to ANN 36%
- Since, data is related to only one ATM, accuracy will be low
- Accuracy of 80+ % can be reached using the same model by having more data for more ATMs

Future Action:

- Collect 4 years data of atlease 100 ATMs located in same geography
- Run multiple models to calculate accuracy