# OSVISIT DATA SET, 81.M

# BY: MEHDI RABBAI

In [56]:
```
#Load libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.seasonal import seasonal_decompose
import statsmodels.tsa.stattools as sts
from statsmodels.graphics.tsaplots import plot_pacf, plot_acf
from statsmodels.tsa.arima_model import ARIMA
from scipy import stats
from sklearn.preprocessing import MinMaxScaler

import seaborn as sns
sns.set()
import warnings
warnings.filterwarnings('ignore')
import statsmodels.api as sm
from sklearn.metrics import mean_squared_error
from math import sqrt
```

In [17]:
```
#Load DATA
Signal = pd.read_csv("C:\\Users\\RABBAI\\Documents\\TP2\\osvisit.csv",header=None)
Signal.head()
```

Out[17]:

| | 0 |
|---|---|
| **0** | 48176 |
| **1** | 35792 |
| **2** | 36376 |
| **3** | 29784 |
| **4** | 21296 |

In [18]:
```
#No null values
Signal[0].isna().value_counts()
```

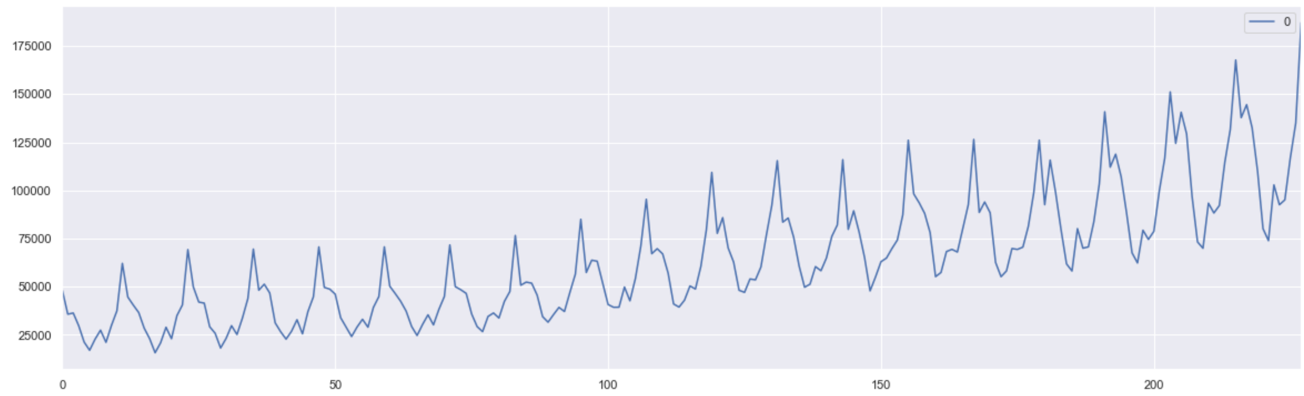
Out[18]:
```
False    228
Name: 0, dtype: int64
```

# Tasks:

### 1. Visualize you time series.

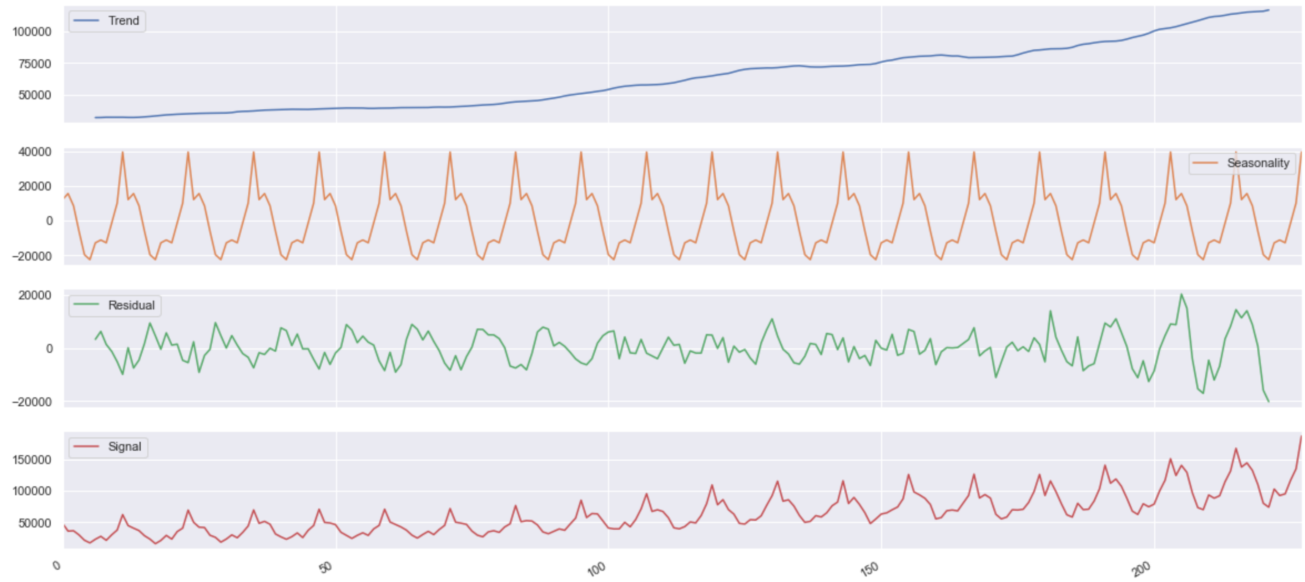
In [19]:
```
#1 Visualize
Signal.plot(figsize=(20,6))
```

Out[19]: <matplotlib.axes._subplots.AxesSubplot at 0x1d779dcff28>




## 2. Extract the time series components.

In [20]:
```
#2 Time series components
additive = seasonal_decompose(Signal[0],model="additive", period=12)
additive_df =pd.concat([additive.trend,additive.seasonal,additive.resid, additive.obs
erved],axis=1)
additive_df.columns=["Trend","Seasonality","Residual","Signal"]
additive_df.plot(subplots=True,figsize=(20,10))
plt.show()
```




## 3. Partition your data into training, validation and testing sets. Justify the used technique for portioning.

**The data is splited using the iloc method to avoid shuffling the data and losing the time series logic**

In [21]:
```
# 3 Partition your data 80:10:10
training_size = int(len(Signal)*0.8)
validation_size = int(len(Signal)*0.1)
testing_size = int(len(Signal)) - validation_size - training_size
```

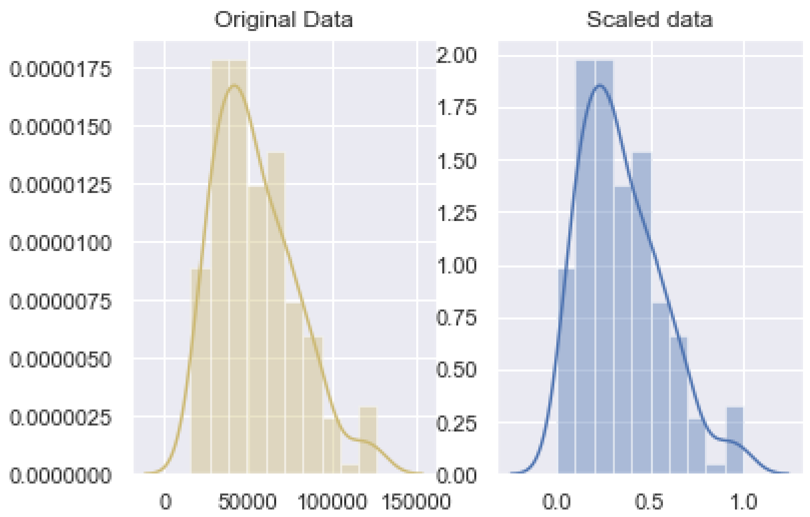
In [22]:
```
Signal_train = Signal.iloc[:training_size]
Signal_val = Signal.iloc[training_size:training_size + validation_size]
Signal_test = Signal.iloc[training_size + validation_size:]
```

### 4. Scale you data using one technique. Justify the usage of data scaling.

**Data scaling allow the machine learning models to learn quickly and to not get baised when trying to estimate the model weights**

In [23]:
```
scaler = MinMaxScaler()
scaler.fit(Signal_train)
Scaled_training_data = scaler.transform(Signal_train)
Scaled_validation_data =scaler.transform(Signal_val)
Scaled_test_data = scaler.transform(Signal_test)
```

In [24]:
```
fig, ax=plt.subplots(1,2)
sns.distplot(Signal_train, ax=ax[0], color='y')
ax[0].set_title("Original Data")
sns.distplot(Scaled_training_data, ax=ax[1])
ax[1].set_title("Scaled data")
plt.show()
```




# I. Statistical modeling:

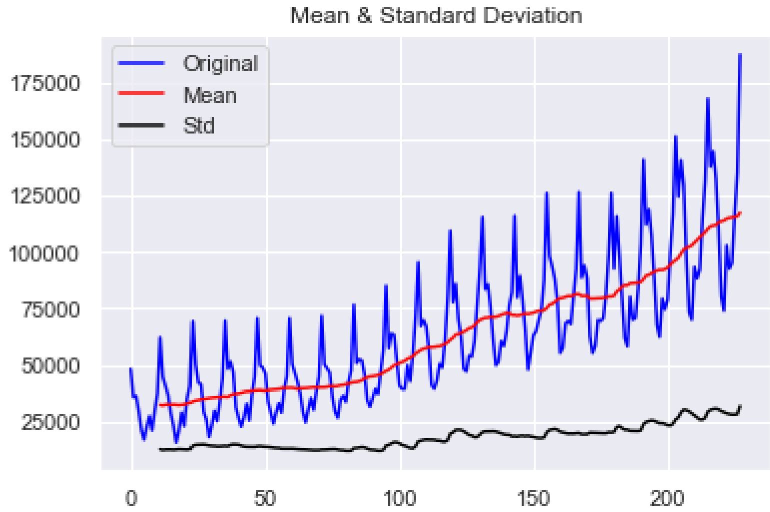
### 1. Is your data stationary? Justify your answer (use graphical interpretation and statistical tests).

In [25]:
```
#1 Data stationary?
adfuller_test_result =sts.adfuller(Signal[0])
print('p-value: ', adfuller_test_result[1])
if adfuller_test_result[1] > 0.05:
    print('The TS is Non-Stationary (p_val > 0.05)')
else:
    print('The TS is Stationary (p_val <= 0.05)')
```

```
p-value:  0.9989422642966919
The TS is Non-Stationary (p_val > 0.05)
```

**The mean and the std is changing through out the time series which makes the data non-stationary also the presence of seasonality makes this time series non-stationary**

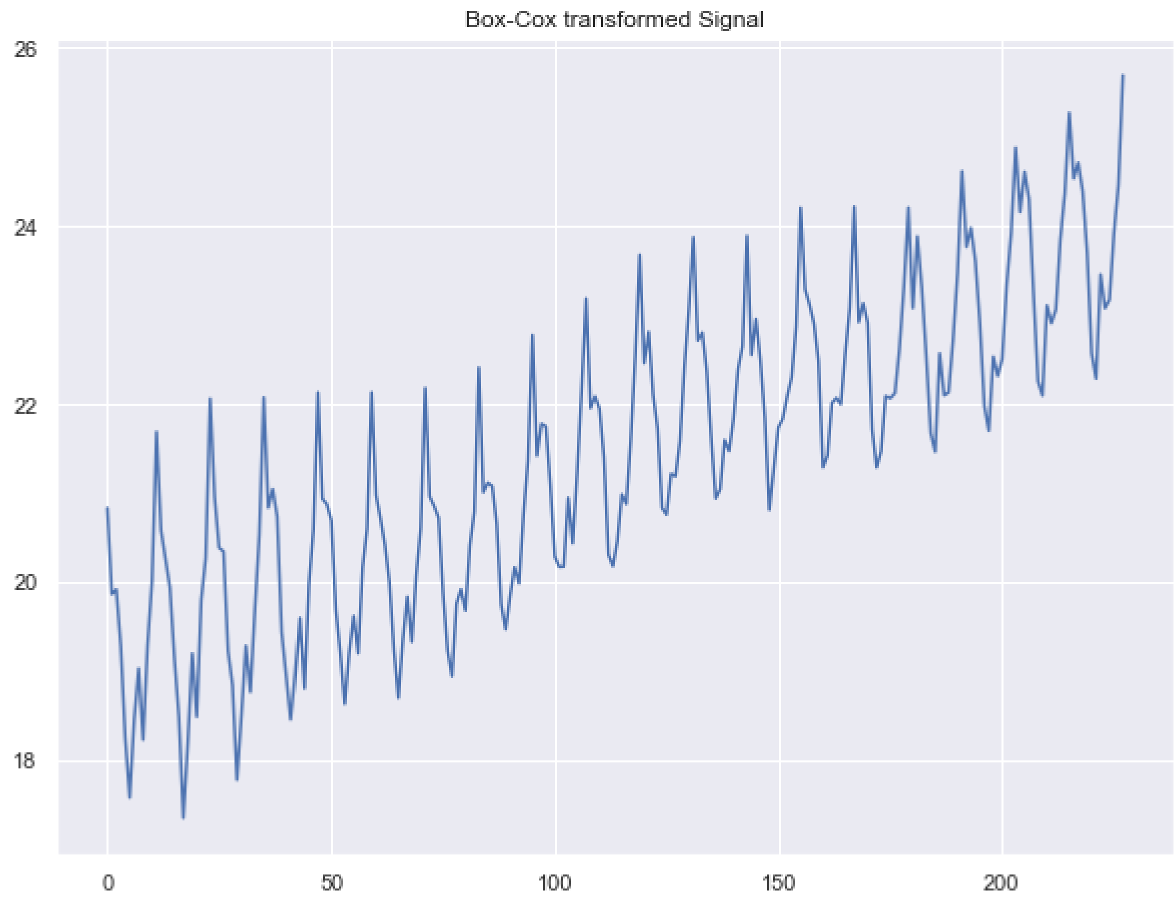
In [15]:
```
mean = Signal.rolling(window = 12).mean()
std = Signal.rolling(window = 12).std()
plt.plot(Signal, color = 'blue', label = 'Original')
plt.plot(mean, color = 'red', label = 'Mean')
plt.plot(std, color = 'black', label = 'Std')
plt.legend(loc = 'best')
plt.title('Mean & Standard Deviation')
plt.show()
```




**2. If not, apply the required transformations and/or differentiations to make it stationary.**

In [26]:
```python
#2 Apply the required transformations and/or differentiations to make it stationary
# Stabilize the variance using the Box-Cox transform
Signal_BC, lmbda = stats.boxcox(Signal[0])
print('the lambda of the Box-Cox transform: ', lmbda)
# Plot TS_BC
plt.figure(figsize=[10, 7.5])
plt.title("Box-Cox transformed Signal")
plt.plot(Signal_BC)
plt.show()
```

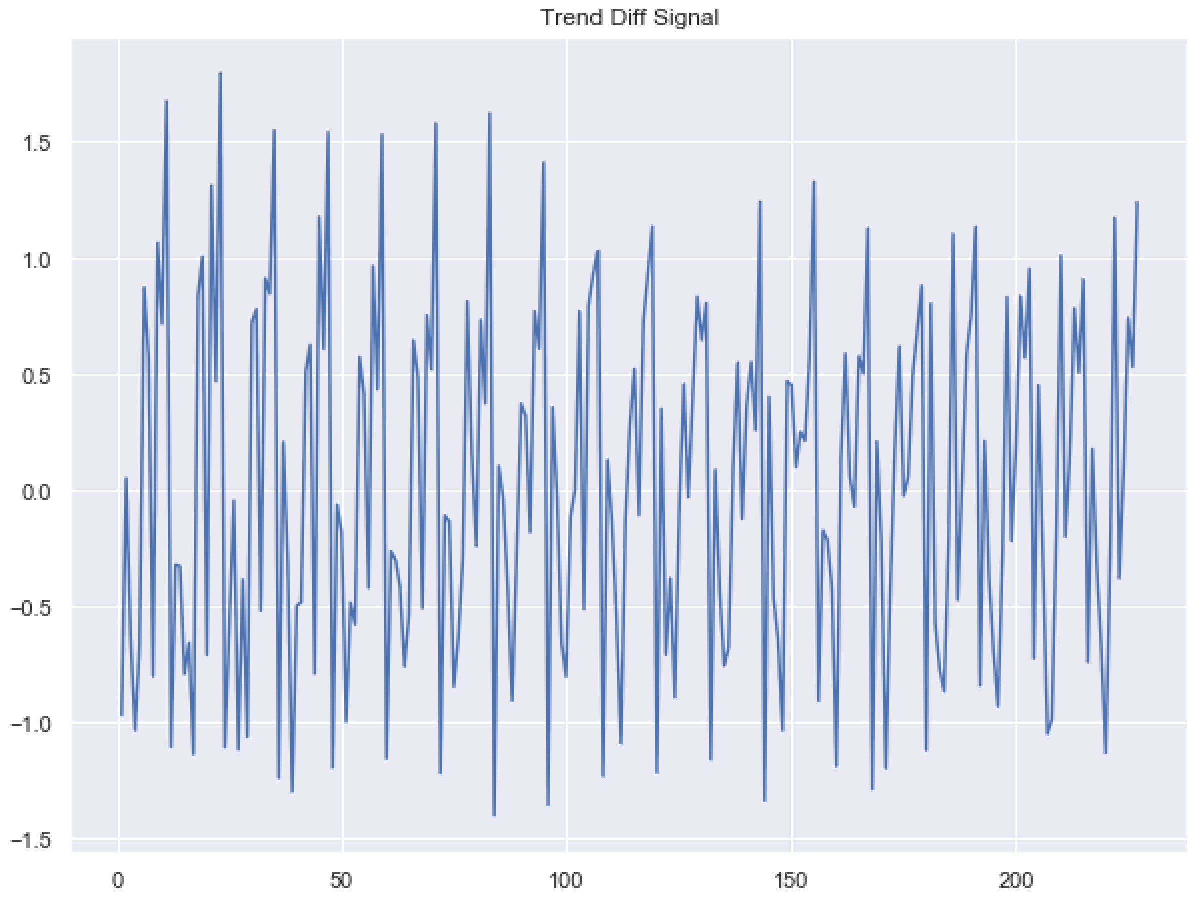
```
the lambda of the Box-Cox transform:  0.11114933445227324
```




In [27]:
```python
adfuller_test_BC_result =sts.adfuller(Signal_BC)
print('p-value: ', adfuller_test_BC_result[1])
if adfuller_test_BC_result[1] > 0.05:
    print('The TS is Non-Stationary (p_val > 0.05)')
else:
    print('The TS is Stationary (p_val <= 0.05)')
```

```
p-value:  0.9816311391521655
The TS is Non-Stationary (p_val > 0.05)
```

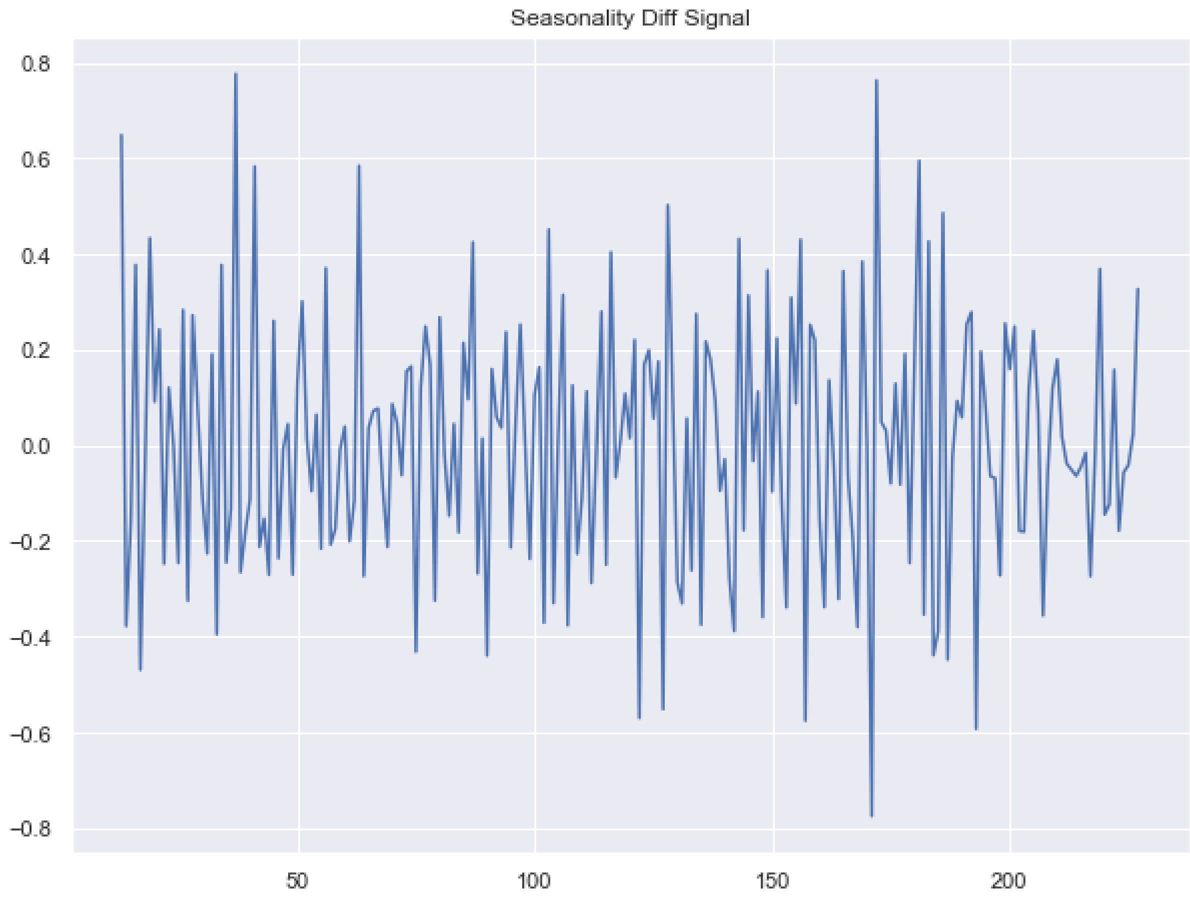
In [28]:
```
# Eliminate the trend
Signal_T_diff = pd.DataFrame(Signal_BC).diff()
Signal_T_diff = Signal_T_diff.drop(Signal_T_diff.index[0])
#Plot_Singal_T_Diff
plt.figure(figsize=[10, 7.5])
plt.title("Trend Diff Signal")
plt.plot(Signal_T_diff)
plt.show()
```




In [29]:
```
adfuller_test_TDIFF_result =sts.adfuller(Signal_T_diff[0])
print('p-value: ', adfuller_test_TDIFF_result[1])
if adfuller_test_TDIFF_result[1] > 0.05:
    print('The TS is Non-Stationary (p_val > 0.05)')
else:
    print('The TS is Stationary (p_val <= 0.05)')
```

```
p-value:  8.788390879171017e-05
The TS is Stationary (p_val <= 0.05)
```

In [30]:
```python
# Eliminate the seasonality
Signal_S_diff = pd.DataFrame(Signal_T_diff).diff(periods=12)
Signal_S_diff = Signal_S_diff.drop(Signal_S_diff.index[0:12])
#Plot_Singal_S_Diff
plt.figure(figsize=[10, 7.5])
plt.title("Seasonality Diff Signal")
plt.plot(Signal_S_diff)
plt.show()
```

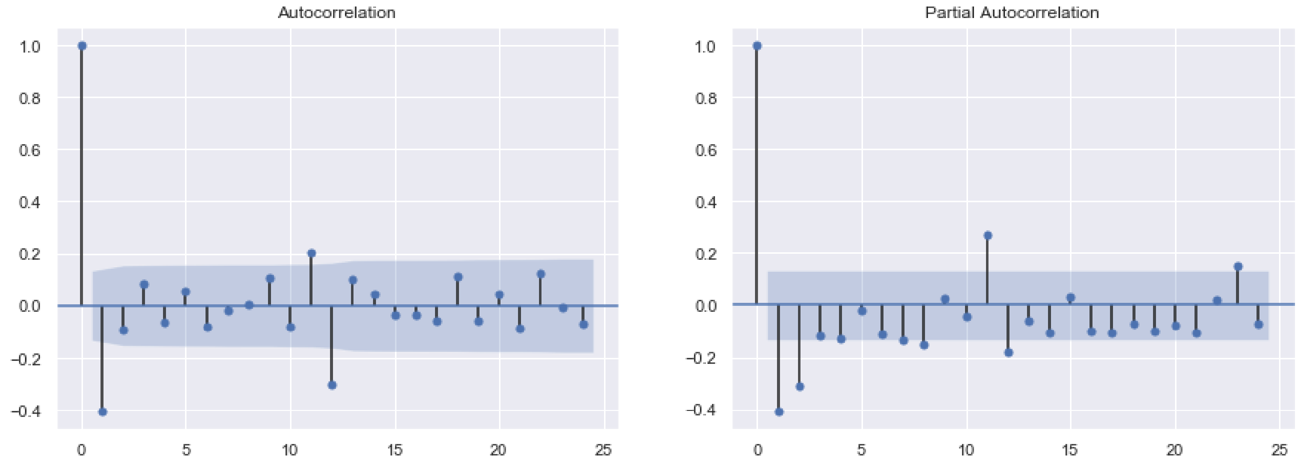



In [31]:
```python
adfuller_test_TSDIFF_result =sts.adfuller(Signal_S_diff[0])
print('p-value: ', adfuller_test_TSDIFF_result[1])
if adfuller_test_TSDIFF_result[1] > 0.05:
    print('The TS is Non-Stationary (p_val > 0.05)')
else:
    print('The TS is Stationary (p_val <= 0.05)')
```

```
p-value:  1.3690536815002095e-05
The TS is Stationary (p_val <= 0.05)
```

### 3. Plot the ACF and PACF of your time series

In [32]:
```
#3 plot acf & pacf
fig, ax = plt.subplots(1,2,figsize=(15,5))
plot_acf(Signal_S_diff,ax[0],);
plot_pacf(Signal_S_diff,ax[1]);
```




## 4. Suggest a set of candidate models.

**The ACF & PACF plots shows that the ACF cuts off after q=1, and PACF cuts off after p=2**

**I will be trying all these models: ARMA(1,1) , ARMA(1,2) , ARMA(2,1) and eventually SARMA(1,1,2)(1,1,2,12)**

## 5. Run each model on your training data and save the training performances.

In [46]:
```
Stationary_Signal = Signal_S_diff.reset_index(drop=True)
Stationary_Signal_train = Stationary_Signal.iloc[:173]
Stationary_Signal_val = Stationary_Signal.iloc[173:173+21]
Stationary_Signal_test = Stationary_Signal.iloc[173+21:]
```

# ARMA(1,1)

In [47]:
```python
model_arma_11 = ARIMA(Stationary_Signal_train, order=(1,0,1))
model_fit_arma_11=model_arma_11.fit()
print(model_fit_arma_11.summary())
```

```
                              ARMA Model Results
==============================================================================
Dep. Variable:                      0   No. Observations:                  173
Model:                     ARMA(1, 1)   Log Likelihood                   4.979
Method:                       css-mle   S.D. of innovations              0.235
Date:                Sun, 31 Jan 2021   AIC                             -1.958
Time:                        04:37:36   BIC                             10.655
Sample:                             0   HQIC                             3.159

==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const          0.0016      0.005      0.321      0.748      -0.008       0.012
ar.L1.0        0.1058      0.144      0.733      0.463      -0.177       0.389
ma.L1.0       -0.7495      0.112     -6.691      0.000      -0.969      -0.530
                                    Roots
=============================================================================
                  Real          Imaginary           Modulus         Frequency
-----------------------------------------------------------------------------
AR.1            9.4498           +0.0000j            9.4498            0.0000
MA.1            1.3341           +0.0000j            1.3341            0.0000
-----------------------------------------------------------------------------
```

# ARMA(1,2)

In [48]:
```python
model_arma_12 = ARIMA(Stationary_Signal_train, order=(1,0,2))
model_fit_arma_12=model_arma_12.fit()
print(model_fit_arma_12.summary())
```

```
                              ARMA Model Results
==============================================================================
Dep. Variable:                      0   No. Observations:                  173
Model:                     ARMA(1, 2)   Log Likelihood                   8.730
Method:                       css-mle   S.D. of innovations              0.228
Date:                Sun, 31 Jan 2021   AIC                             -7.460
Time:                        04:37:54   BIC                              8.306
Sample:                             0   HQIC                            -1.064

==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const          0.0017      0.006      0.313      0.754      -0.009       0.013
ar.L1.0       -0.8814      0.039    -22.319      0.000      -0.959      -0.804
ma.L1.0        0.2968      0.069      4.309      0.000       0.162       0.432
ma.L2.0       -0.7032      0.067    -10.553      0.000      -0.834      -0.573
                                    Roots
=============================================================================
                  Real          Imaginary           Modulus         Frequency
-----------------------------------------------------------------------------
AR.1           -1.1346           +0.0000j            1.1346            0.5000
MA.1           -1.0000           +0.0000j            1.0000            0.5000
MA.2            1.4220           +0.0000j            1.4220            0.0000
-----------------------------------------------------------------------------
```

# ARMA(2,1)

In [49]:
```python
model_arma_21 = ARIMA(Stationary_Signal_train, order=(2,0,1))
model_fit_arma_21=model_arma_21.fit()
print(model_fit_arma_21.summary())
```

```
                              ARMA Model Results
==============================================================================
Dep. Variable:                      0   No. Observations:                  173
Model:                     ARMA(2, 1)   Log Likelihood                   5.851
Method:                       css-mle   S.D. of innovations              0.233
Date:                Sun, 31 Jan 2021   AIC                             -1.702
Time:                        04:39:38   BIC                             14.065
Sample:                             0   HQIC                             4.695

==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const          0.0004      0.002      0.174      0.862      -0.004       0.004
ar.L1.0        0.2955      0.100      2.964      0.003       0.100       0.491
ar.L2.0        0.1754      0.096      1.824      0.068      -0.013       0.364
ma.L1.0       -0.9442      0.061    -15.355      0.000      -1.065      -0.824
                                    Roots
=============================================================================
                  Real          Imaginary           Modulus         Frequency
-----------------------------------------------------------------------------
AR.1            1.6897           +0.0000j            1.6897            0.0000
AR.2           -3.3746           +0.0000j            3.3746            0.5000
MA.1            1.0591           +0.0000j            1.0591            0.0000
-----------------------------------------------------------------------------
```

## SARIMA(1,1,2)(1,1,2,12)

In [78]:
```
model=sm.tsa.statespace.SARIMAX(Signal_train,orders=(1, 1, 2),seasonal_order=(1,1,2,1
2))
model_fit=model.fit()
print(model_fit.summary())
```

```
                                      SARIMAX Results
==========================================================================================
Dep. Variable:                                     0   No. Observations:
182
Model:             SARIMAX(1, 0, 0)x(1, 1, [1, 2], 12)   Log Likelihood
-1675.885
Date:                               Sun, 31 Jan 2021   AIC
3361.770
Time:                                       05:07:47   BIC
3377.449
Sample:                                            0   HQIC
3368.133
                                               - 182
Covariance Type:                                 opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          0.2625      0.034      7.610      0.000       0.195       0.330
ar.S.L12       0.9993      0.010    101.248      0.000       0.980       1.019
ma.S.L12      -1.1329      0.078    -14.535      0.000      -1.286      -0.980
ma.S.L24       0.1444      0.047      3.061      0.002       0.052       0.237
sigma2      1.972e+07   4.11e-09    4.8e+15      0.000    1.97e+07    1.97e+07
===================================================================================
Ljung-Box (L1) (Q):                   4.70   Jarque-Bera (JB):                63.22
Prob(Q):                              0.03   Prob(JB):                         0.00
Heteroskedasticity (H):               2.43   Skew:                            -0.19
Prob(H) (two-sided):                  0.00   Kurtosis:                         5.96
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-ste
p).
[2] Covariance matrix is singular or near-singular, with condition number 1.66e+30.
Standard errors may be unstable.
```

## 6. Select the best model based on validation performance.

In [183]:
```
ARMA_11_val_pred = model_fit_arma_11.predict(start=len(Stationary_Signal_train),end=len(Stationary_Signal_val)+len(Stationary_Signal_train)-1,dynamic=False)

RMSE_val_ARMA_11 = sqrt(mean_squared_error(Stationary_Signal_val, ARMA_11_val_pred))
MAE_11 = mean_absolute_error(Stationary_Signal_val, ARMA_11_val_pred)

print("ARMA(1,1): VAL RMSE ERROR  ",round(RMSE_val_ARMA_11,2)," VAL MAE ERROR ", round(MAE_11,2))


ARMA_12_val_pred = model_fit_arma_12.predict(start=len(Stationary_Signal_train),end=len(Stationary_Signal_val)+len(Stationary_Signal_train)-1,dynamic=False)

RMSE_val_ARMA_12 = sqrt(mean_squared_error(Stationary_Signal_val, ARMA_12_val_pred))
MAE_12 = mean_absolute_error(Stationary_Signal_val, ARMA_12_val_pred)

print("ARMA(1,2): VAL RMSE ERROR  ",round(RMSE_val_ARMA_12,2)," VAL MAE ERROR ", round(MAE_12,2))

ARMA_21_val_pred = model_fit_arma_21.predict(start=len(Stationary_Signal_train),end=len(Stationary_Signal_val)+len(Stationary_Signal_train)-1,dynamic=False)

RMSE_val_ARMA_21 = sqrt(mean_squared_error(Stationary_Signal_val, ARMA_21_val_pred))
MAE_21 = mean_absolute_error(Stationary_Signal_val, ARMA_21_val_pred)

print("ARMA(2,1): VAL RMSE ERROR  ",round(RMSE_val_ARMA_21,2)," VAL MAE ERROR ", round(MAE_21,2))
```

```
ARMA(1,1): VAL RMSE ERROR   0.24  VAL MAE ERROR  0.2
ARMA(1,2): VAL RMSE ERROR   0.25  VAL MAE ERROR  0.2
ARMA(2,1): VAL RMSE ERROR   0.24  VAL MAE ERROR  0.2
```

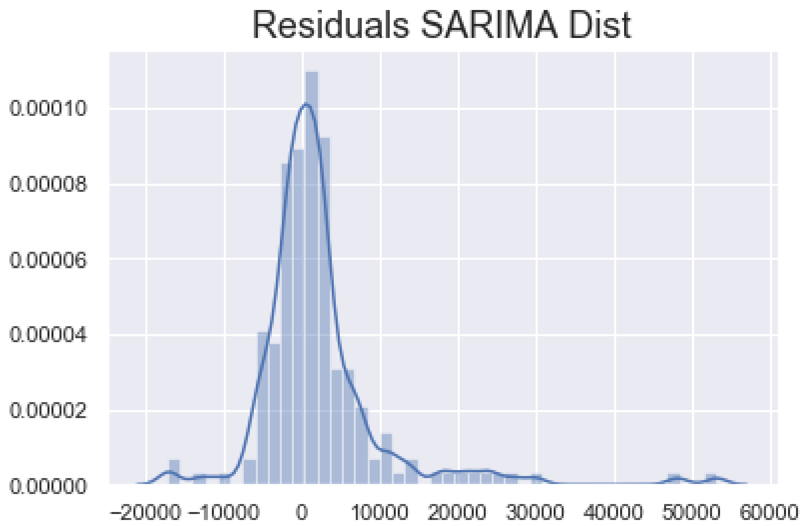
**Because the validation performance of the 3 models is quite similar, I'll be choosing the model with the lowest AIC and the most significant coefficients ARMA(1,2)**

**Eventually the final model that will be used to model the original series is going to be a SARIMA Model because of the presence of seasonality & the non-stationarity of the TIME SERIES experimentally this one gives better results & follow the under laying logic of the DATA**

## 7. Evaluate the errors of your model using residual analysis.

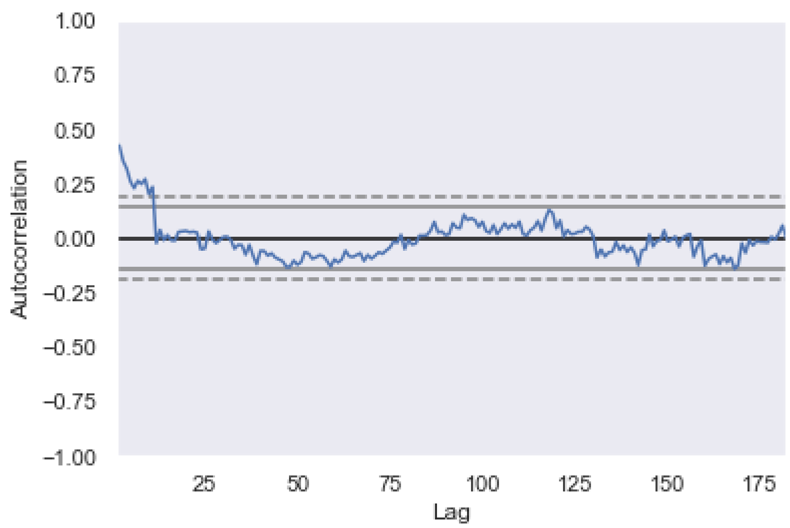
**The residuals follows a gauss distribution**

In [188]:
```
sns.distplot(model_fit.resid)
plt.title("Residuals SARIMA Dist",size = 18)
plt.show()
```




## 8. Are your residuals white noise? (Use graphical interpretation and statistical tests).

In [185]:
```
from pandas.plotting import autocorrelation_plot

autocorrelation_plot(model_fit.resid)
plt.show()
```




## Ljung_Box White Noise Test

**Null Hypothesis : There is no autocorrelation between the signal and its lagged version**

**Alternate Hypothesis : There is significant autocorrelation between the signal and its lagged version**

In [190]:
```
from statsmodels.stats.diagnostic import acorr_ljungbox
acorr_ljungbox(model_fit.resid,period=12,return_df=True)
```

Out[190]:

| | lb_stat | lb_pvalue |
|---|---|---|
| **1** | 34.370945 | 4.554721e-09 |
| **2** | 67.754891 | 1.937369e-15 |
| **3** | 91.194092 | 1.213569e-19 |
| **4** | 110.239881 | 6.468911e-23 |
| **5** | 122.917721 | 7.560534e-25 |
| **6** | 132.834643 | 3.250366e-26 |
| **7** | 146.010872 | 2.791734e-28 |
| **8** | 157.790662 | 4.632462e-30 |
| **9** | 171.941368 | 2.431203e-32 |
| **10** | 179.818445 | 2.555414e-33 |
| **11** | 190.564670 | 6.712525e-35 |
| **12** | 190.679756 | 2.721097e-34 |
| **13** | 190.947913 | 9.813298e-34 |
| **14** | 190.969256 | 3.816703e-33 |
| **15** | 191.004206 | 1.419645e-32 |
| **16** | 191.028480 | 5.121713e-32 |
| **17** | 191.061156 | 1.780614e-31 |
| **18** | 191.220459 | 5.662749e-31 |
| **19** | 191.408489 | 1.727416e-30 |
| **20** | 191.621475 | 5.072535e-30 |
| **21** | 191.761303 | 1.501223e-29 |
| **22** | 191.933333 | 4.274301e-29 |
| **23** | 192.057806 | 1.214937e-28 |
| **24** | 192.647186 | 2.748674e-28 |

### Since the p-value < 0.5 we reject the Null Hypothesis thus There is significant autocorrelation between the signal and its lagged version (the residuals are not a white noise series)

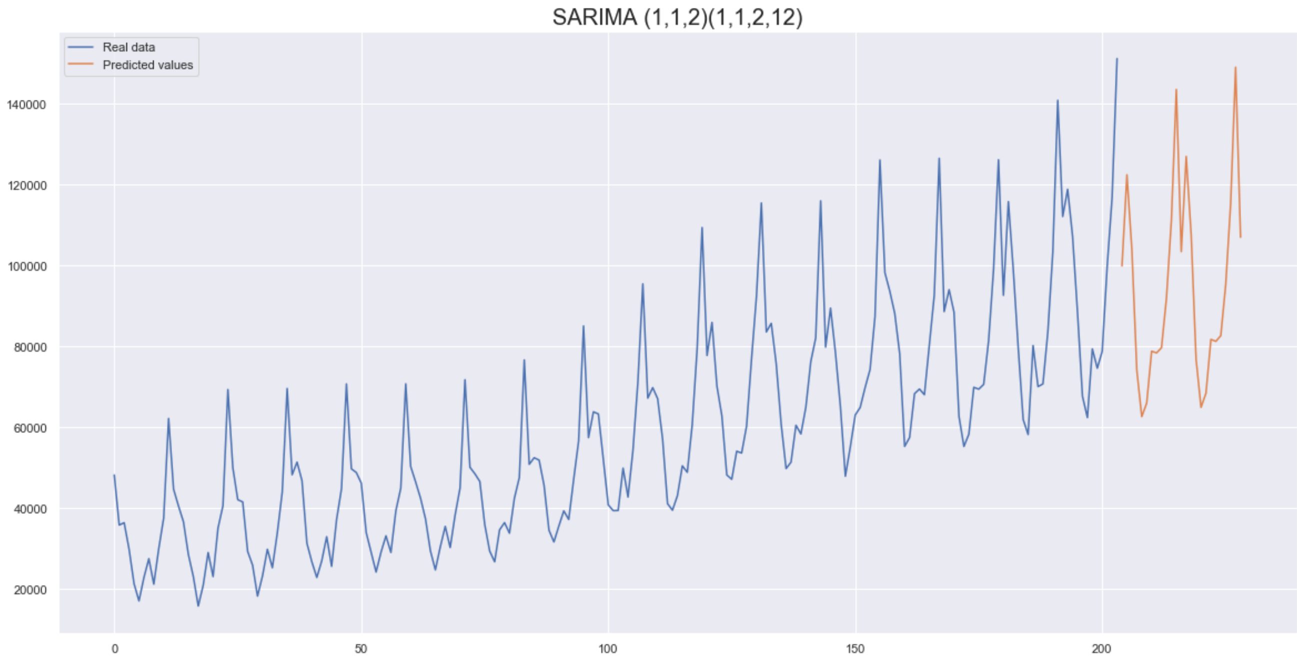
## 9. If not, model your residuals using heteroskedastic models.

In [191]:
```
import arch
garch_model = arch.arch_model(model_fit.resid,p=1,q=2)
garch_fitted = garch_model.fit()
```

```
Iteration:      1,   Func. Count:      7,   Neg. LLF: 6002.701275285661
Iteration:      2,   Func. Count:     15,   Neg. LLF: 1830.0598592585484
Iteration:      3,   Func. Count:     21,   Neg. LLF: 1829.4446026796486
Iteration:      4,   Func. Count:     27,   Neg. LLF: 1829.4037075721599
Iteration:      5,   Func. Count:     33,   Neg. LLF: 1829.399337932935
Iteration:      6,   Func. Count:     39,   Neg. LLF: 1829.3990613167616
Iteration:      7,   Func. Count:     45,   Neg. LLF: 1829.3989195653467
Iteration:      8,   Func. Count:     51,   Neg. LLF: 1829.398557924079
Iteration:      9,   Func. Count:     57,   Neg. LLF: 1829.3976267847797
Iteration:     10,   Func. Count:     63,   Neg. LLF: 1829.3950340940705
Iteration:     11,   Func. Count:     69,   Neg. LLF: 1829.3887782500465
Iteration:     12,   Func. Count:     75,   Neg. LLF: 1829.370550131861
Iteration:     13,   Func. Count:     81,   Neg. LLF: 1829.3278198824469
Iteration:     14,   Func. Count:     87,   Neg. LLF: 1829.198704203607
Iteration:     15,   Func. Count:     93,   Neg. LLF: 1828.8929254128234
Iteration:     16,   Func. Count:     99,   Neg. LLF: 1828.1376354192528
Iteration:     17,   Func. Count:    105,   Neg. LLF: 1826.0636288400124
Iteration:     18,   Func. Count:    111,   Neg. LLF: 1821.682886225136
Iteration:     19,   Func. Count:    117,   Neg. LLF: 1820.0980909654645
Iteration:     20,   Func. Count:    123,   Neg. LLF: 1818.677956166246
Iteration:     21,   Func. Count:    129,   Neg. LLF: 1818.5156902740064
Iteration:     22,   Func. Count:    135,   Neg. LLF: 1818.4941156869752
Iteration:     23,   Func. Count:    141,   Neg. LLF: 1818.4936215082794
Iteration:     24,   Func. Count:    147,   Neg. LLF: 1818.4936199798578
Iteration:     25,   Func. Count:    153,   Neg. LLF: 1818.543772702622
Optimization terminated successfully    (Exit mode 0)
            Current function value: 1818.4936198728826
            Iterations: 25
            Function evaluations: 158
            Gradient evaluations: 25
```

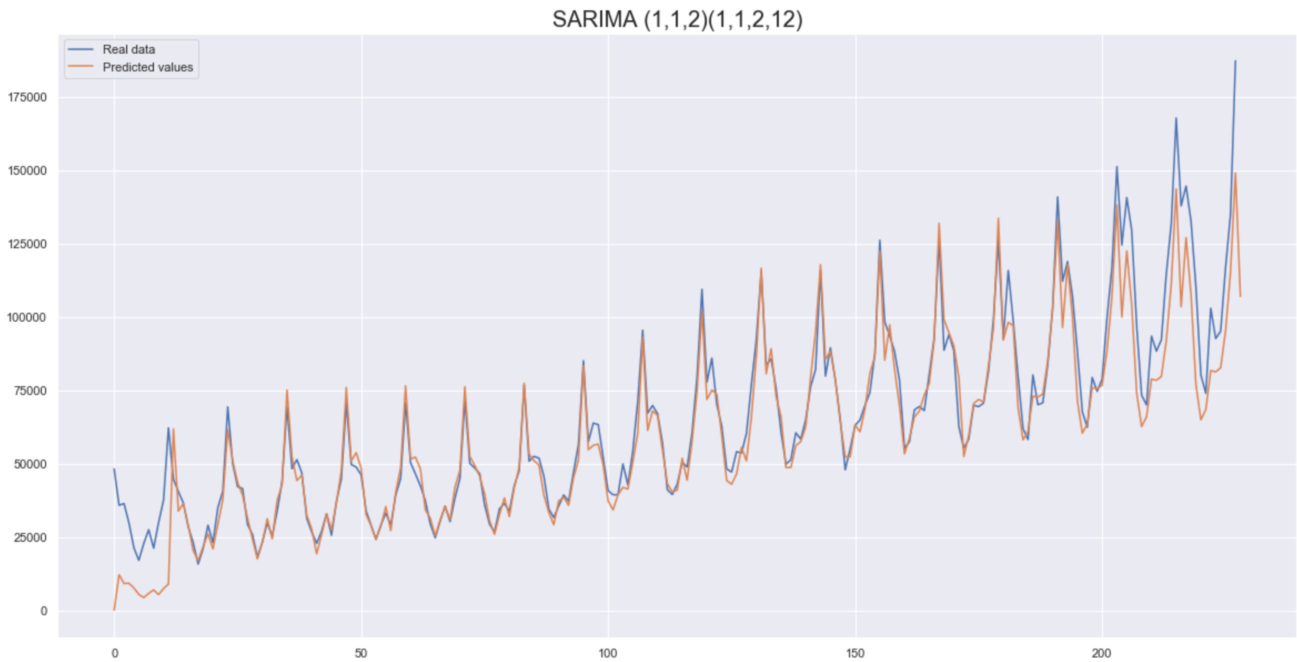
## 10. Generate forecasts based on the selected model.

```
In [195]: plt.figure(figsize=(20,10))
df_pred = model_fit.predict(start=training_size+validation_size,end=validation_size+training_size+testing_size,dynamic=False)
plt.plot(Signal[:training_size+validation_size],label="Real data")
plt.plot(df_pred,label="Predicted values")
plt.title("SARIMA (1,1,2)(1,1,2,12)",size=20)
plt.legend()
plt.show()
```




# 11. Plot the real and estimated training, validation and testing data in one graph

```
In [208]: plt.figure(figsize=(20,10))
df_pred = model_fit.predict(start=0,end=validation_size+training_size+testing_size,dynamic=False)
plt.plot(Signal,label="Real data")
plt.plot(df_pred,label="Predicted values")
plt.title("SARIMA (1,1,2)(1,1,2,12)",size=20)
plt.legend()
plt.show()
```



## 12. Provide the training and testing error for your model.

In [210]:

```
predicted_ = model_fit.predict(start = 0, end = Signal.shape[0])
predicted_training_ = predicted.iloc[:training_size]
predicted_test_ = predicted.iloc[-testing_size:]
MAE_train_ = mean_absolute_error(Signal_train, predicted_training_)
MAE_test_ = mean_absolute_error(Signal_test, predicted_test_)

print(" TRAIN MAE ERROR  ",round(MAE_train_,2)," TEST MAE ERROR ", round(MAE_test_,2
))
```

```
 TRAIN MAE ERROR   54346.21  TEST MAE ERROR  113806.67
```

# OSVISIT DATA SET, 81.M

# BY: MEHDI RABBAI

In [1]:
```
import pandas as pd
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')
```

In [2]:
```
df = pd.read_csv('C:\\Users\\RABBAI\\Documents\\TP2\\osvisit.csv',header=None)
```

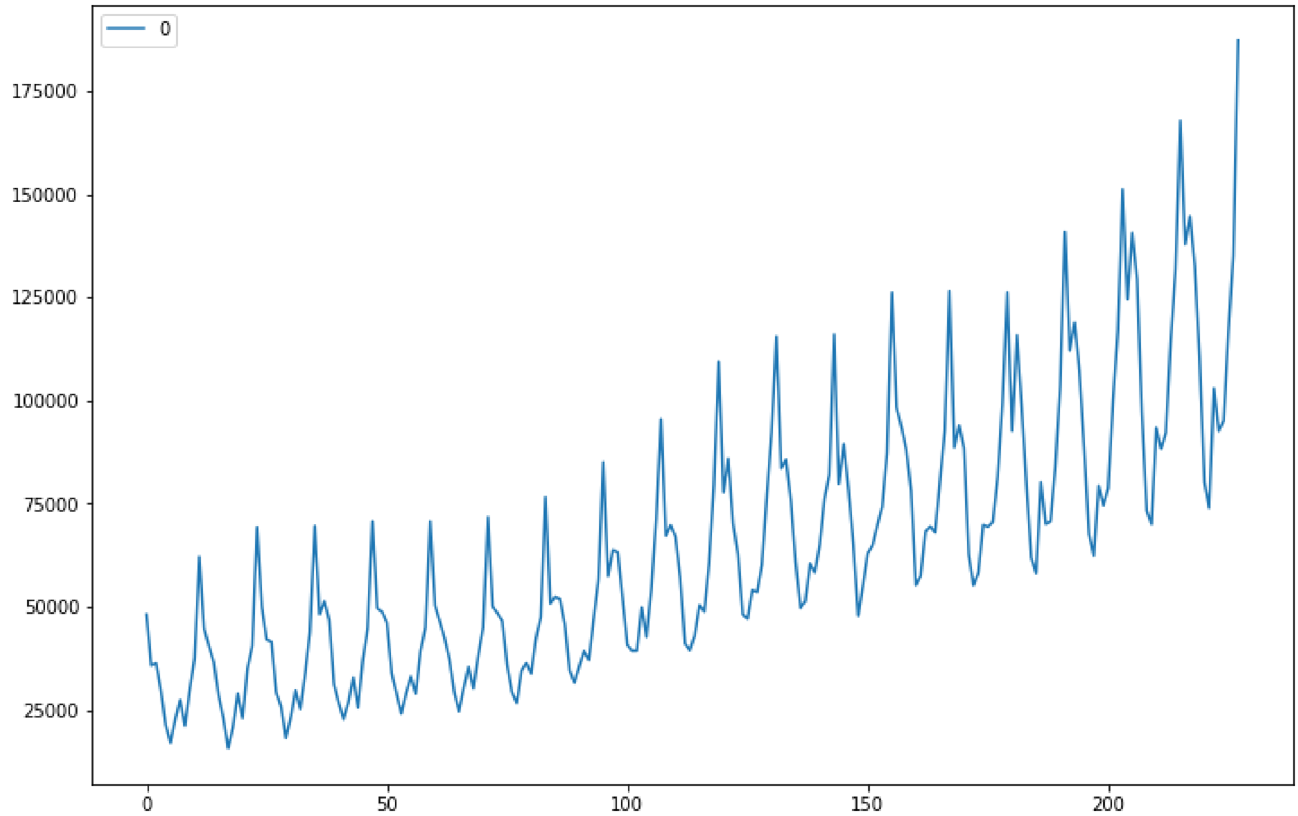
In [3]:
```
df.head()
```

Out[3]:

| | 0 |
|---|---|
| 0 | 48176 |
| 1 | 35792 |
| 2 | 36376 |
| 3 | 29784 |
| 4 | 21296 |

In [4]:
```
df.plot(figsize=(12,8))
```

Out[4]: <matplotlib.axes._subplots.AxesSubplot at 0x17d7c6ad160>




# Preprocessing

In [5]:
```python
len(df)
train = df.iloc[:216]
test = df.iloc[216:]
```

In [6]:
```python
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
scaler.fit(train)
scaled_train = scaler.transform(train)
scaled_test = scaler.transform(test)
```

# II. Machine learning modeling:

### 1. Select one machine learning model from your choice to model the given time series.

**The model that will be used to model the data is RNN (LSTM)**

### 2. Justify the choice of your model. And prove that this model can be used for time series forecasting.

**Long Short-Term Memory (LSTM) is a type of recurrent neural network that can learn the order dependence between items in a sequence. LSTMs have the promise of being able to learn the context required to make predictions in time series forecasting problems, rather than having this context pre-specified and fixed**

### 3. Provide the form of the hypothesis set of your model.

### 4. Define the parameters and the hyper-parameters of your model.

**Creating a time series generator**

In [7]:
```python
def generator_to_array(generator):
    """ Function that allow to extract X_train & y_train from a time series generator
    (helps us in validation since .fit function doesnt allow validation.split from a
 time series generator)"""
    X_train=list()
    y_train=list()
    for i in range(len(generator)):
        x=generator[i][0].flatten().tolist()
        y=generator[i][1].flatten().tolist()
        X_train.append(x)
        y_train.append(y)
    return np.array(X_train), np.array(y_train)
```

In [8]:
```
from keras.preprocessing.sequence import TimeseriesGenerator
n_input = 12
n_features = 1
generator = TimeseriesGenerator(scaled_train, scaled_train, length=n_input, batch_siz
e=1)
```

In [9]:
```
# What does the first batch look like?
X,y = generator[0]
print(f'Given the Array: \n{X.flatten()}')
print(f'Predict this y: \n {y}')
```

```
Given the Array:
[0.21332631 0.1318285  0.13567174 0.09229048 0.03643184 0.0083709
 0.04635583 0.07710177 0.03558948 0.09323813 0.14317397 0.30532724]
Predict this y:
 [[0.19026686]]
```

**Creating the model**

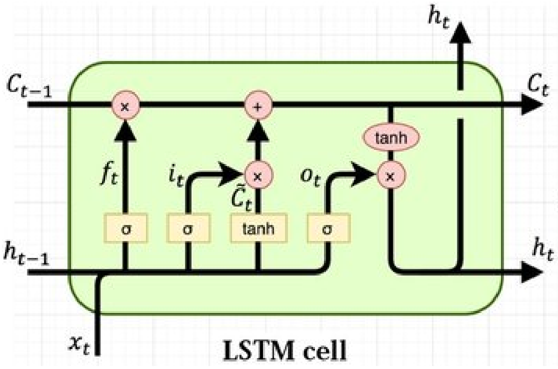
In [10]:
```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
import keras
from keras.layers import Dropout
```

In [11]:
```
X_train, y_train = generator_to_array(generator)
X_train  = X_train.reshape(204,12,1)
X_train.shape
```

Out[11]:
```
(204, 12, 1)
```

In [71]:
```
model = keras.Sequential()
model.add(
  keras.layers.Bidirectional(
    keras.layers.LSTM(
      units=100,
      input_shape=(X_train.shape[1], X_train.shape[2])
    )
  )
)
#model.add(keras.layers.Dropout(rate=0.2))
model.add(keras.layers.Dense(units=1))
model.compile(loss='mean_squared_error', optimizer='adam')
```

## 5. Explain the functioning of the selected model using mathematical equations and graphs.




$$i_t = \sigma(x_t U^i + h_{t-1} W^i)$$
$$f_t = \sigma(x_t U^f + h_{t-1} W^f)$$
$$o_t = \sigma(x_t U^o + h_{t-1} W^o)$$
$$\tilde{C}_t = \tanh(x_t U^g + h_{t-1} W^g)$$
$$C_t = \sigma(f_t * C_{t-1} + i_t * \tilde{C}_t)$$
$$h_t = \tanh(C_t) * o_t$$

## 6. Provide a set of candidate models. Justify your choice.

***This model has no dropout***

In [ ]:
```
model = keras.Sequential()
model.add(
  keras.layers.Bidirectional(
    keras.layers.LSTM(
      units=100,
      input_shape=(X_train.shape[1], X_train.shape[2])
    )
  )
)
model.add(keras.layers.Dense(units=1))
model.compile(loss='mean_squared_error', optimizer='adam')
```

***This model has dropout which is a regulator that can lead to better results (chosen)***

In [ ]:
```
model = keras.Sequential()
model.add(
  keras.layers.Bidirectional(
    keras.layers.LSTM(
      units=100,
      input_shape=(X_train.shape[1], X_train.shape[2])
    )
  )
)
model.add(keras.layers.Dropout(rate=0.2))
model.add(keras.layers.Dense(units=1))
model.compile(loss='mean_squared_error', optimizer='adam')
```

**we can also change the number of nodes in LSTM layer which can lead to a more complex model but not necessary better results**

## 7. Choose one learning algorithm to minimize your cost function. Justify your choice.

***a) Select a learning rate of your algorithm (constant or adaptive learning rate). Justify your choice. If it is adaptive provide the adaption technique.***

***The learning rate is adaptive, since this one provides better results & adapts to the data, the optimizer used is ADAM (adaptive moment estimation optimizer)***

***b) Fix the parameters of your learning algorithm.***

## 8. Use your models to learn the parameters based on training data, and save the training performances.

In [113]:

```
ES = keras.callbacks.EarlyStopping(patience = 5)
history = model.fit(
    X_train, y_train,
    epochs=30,
    batch_size=1,
    validation_split=0.1,
    shuffle=False,

)
```

```
Epoch 1/30
183/183 [==============================] - 1s 7ms/step - loss: 0.0012 - val_loss: 0.
0014
Epoch 2/30
183/183 [==============================] - 1s 7ms/step - loss: 0.0012 - val_loss: 0.
0014
Epoch 3/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0012 - val_loss: 0.
0014
Epoch 4/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0012 - val_loss: 0.
0014
Epoch 5/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0012 - val_loss: 0.
0014
Epoch 6/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0012 - val_loss: 0.
0014
Epoch 7/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0012 - val_loss: 0.
0015
Epoch 8/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0012 - val_loss: 0.
0015
Epoch 9/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0012 - val_loss: 0.
0015
Epoch 10/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0012 - val_loss: 0.
0015
Epoch 11/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0012 - val_loss: 0.
0015
Epoch 12/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0012 - val_loss: 0.
0015
Epoch 13/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0012 - val_loss: 0.
0015
Epoch 14/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0012 - val_loss: 0.
0015
Epoch 15/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0012 - val_loss: 0.
0015
Epoch 16/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0012 - val_loss: 0.
0015
Epoch 17/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0012 - val_loss: 0.
0015
Epoch 18/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0012 - val_loss: 0.
0015
Epoch 19/30
183/183 [==============================] - 1s 7ms/step - loss: 0.0012 - val_loss: 0.
0015
Epoch 20/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0012 - val_loss: 0.
0016
Epoch 21/30
183/183 [==============================] - 1s 7ms/step - loss: 0.0012 - val_loss: 0.
0016
Epoch 22/30
183/183 [==============================] - 1s 7ms/step - loss: 0.0012 - val_loss: 0.
```

```
0015
Epoch 23/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0011 - val_loss: 0.
0016
Epoch 24/30
183/183 [==============================] - 1s 8ms/step - loss: 0.0011 - val_loss: 0.
0016
Epoch 25/30
183/183 [==============================] - 1s 7ms/step - loss: 0.0011 - val_loss: 0.
0016
Epoch 26/30
183/183 [==============================] - 1s 7ms/step - loss: 0.0011 - val_loss: 0.
0016
Epoch 27/30
183/183 [==============================] - 1s 7ms/step - loss: 0.0011 - val_loss: 0.
0016
Epoch 28/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0011 - val_loss: 0.
0016
Epoch 29/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0011 - val_loss: 0.
0016
Epoch 30/30
183/183 [==============================] - 1s 7ms/step - loss: 0.0011 - val_loss: 0.
0016
```

In [96]:

```
model2 = keras.Sequential()
model2.add(
  keras.layers.Bidirectional(
    keras.layers.LSTM(
      units=100,
      input_shape=(X_train.shape[1], X_train.shape[2])
    )
  )

)
model2.add(keras.layers.Dropout(rate=0.2))
model2.add(keras.layers.Dense(units=1))
model2.compile(loss='mean_squared_error', optimizer='adam')
```

In [114]:

```
ES = keras.callbacks.EarlyStopping(patience = 5)
history = model2.fit(
    X_train, y_train,
    epochs=30,
    batch_size=1,
    validation_split=0.1,
    shuffle=False,
)
```

```
Epoch 1/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0018 - val_loss: 0.
0014
Epoch 2/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0018 - val_loss: 0.
0016
Epoch 3/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0021 - val_loss: 0.
0018
Epoch 4/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0015 - val_loss: 0.
0015
Epoch 5/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0015 - val_loss: 0.
0014
Epoch 6/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0016 - val_loss: 0.
0015
Epoch 7/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0017 - val_loss: 0.
0014
Epoch 8/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0018 - val_loss: 0.
0018
Epoch 9/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0015 - val_loss: 0.
0012
Epoch 10/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0018 - val_loss: 0.
0014
Epoch 11/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0016 - val_loss: 0.
0012
Epoch 12/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0019 - val_loss: 0.
0015
Epoch 13/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0016 - val_loss: 0.
0012
Epoch 14/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0016 - val_loss: 0.
0015
Epoch 15/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0015 - val_loss: 0.
0015
Epoch 16/30
183/183 [==============================] - 1s 7ms/step - loss: 0.0015 - val_loss: 0.
0012
Epoch 17/30
183/183 [==============================] - 1s 8ms/step - loss: 0.0015 - val_loss: 0.
0016
Epoch 18/30
183/183 [==============================] - 1s 7ms/step - loss: 0.0015 - val_loss: 0.
0016
Epoch 19/30
183/183 [==============================] - 1s 7ms/step - loss: 0.0018 - val_loss: 0.
0012.91
Epoch 20/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0016 - val_loss: 0.
0015
Epoch 21/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0015 - val_loss: 0.
0013
Epoch 22/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0014 - val_loss: 0.
```

```
0014
Epoch 23/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0016 - val_loss: 0.
0019
Epoch 24/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0017 - val_loss: 0.
0016
Epoch 25/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0016 - val_loss: 0.
0020
Epoch 26/30
183/183 [==============================] - 1s 7ms/step - loss: 0.0012 - val_loss: 0.
0013
Epoch 27/30
183/183 [==============================] - 1s 7ms/step - loss: 0.0018 - val_loss: 0.
0018
Epoch 28/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0015 - val_loss: 0.
0020
Epoch 29/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0016 - val_loss: 0.
0018
Epoch 30/30
183/183 [==============================] - 1s 6ms/step - loss: 0.0015 - val_loss: 0.
0022
```

### 9. Select the best model using the validation data.

In [115]:

```
print("Model 1 Val loss = ", model.history.history["val_loss"][-1])
print("Model 2 Val loss = ", model2.history.history["val_loss"][-1])
```

```
Model 1 Val loss =  0.0016260950360447168
Model 2 Val loss =  0.002159732626751065З
```

### We'll continue using Model 1

### 10. Test your selected model on the testing data and retain your results based on the statistical metrics.

In [116]:

```
test_predictions = []

first_eval_batch = scaled_train[-n_input:]
current_batch = first_eval_batch.reshape((1, n_input, n_features))

for i in range(len(test)):

    # get prediction 1 time stamp ahead ([0] is for grabbing just the number instead
 of [array])
    current_pred = model.predict(current_batch)[0]

    # store prediction
    test_predictions.append(current_pred)

    # update batch to now include prediction and drop first value
    current_batch = np.append(current_batch[:,1:,:],[[current_pred]],axis=1)
```

In [117]:
```
true_predictions = scaler.inverse_transform(test_predictions)
test['Predictions'] = true_predictions
```

In [118]:
```
import math
from sklearn.metrics import mean_squared_error
# Calculate root mean squared error
# trainScore = math.sqrt(mean_squared_error(original_train[-len(true_train_prediction
s):], true_train_predictions))
# print('M2 Train Score: %.2f RMSE' % (trainScore))
# trainScore = math.sqrt(mean_squared_error(original_train[-len(true_train_prediction
s_):], true_train_predictions_))
# print('m1 Test Score: %.2f RMSE' % (trainScore))
#testScore = math.sqrt(mean_squared_error(test[0],test_["Predictions"]))
#print('M2 Train Score: %.2f RMSE' % (testScore))
testScore = math.sqrt(mean_squared_error(test[0],test["Predictions"]))
print('M1 Test Score: %.2f RMSE' % (testScore))
```

```
M1 Test Score: 5838.17 RMSE
```

## 11. Is there any overfitting or underfitting problem?

**The validation loss dont increase at the end which indicates the absence of overfitting**

## 12. Propose a solution to fix the problem of overfitting?

**In case of overfitting, Early stopping can be usedwhen fitting the model to avoid overfitting**

## 13. If there is any overfitting problem try to fix it using the proposed solution. And compare the performance before and after applying the solution. What can you notice?

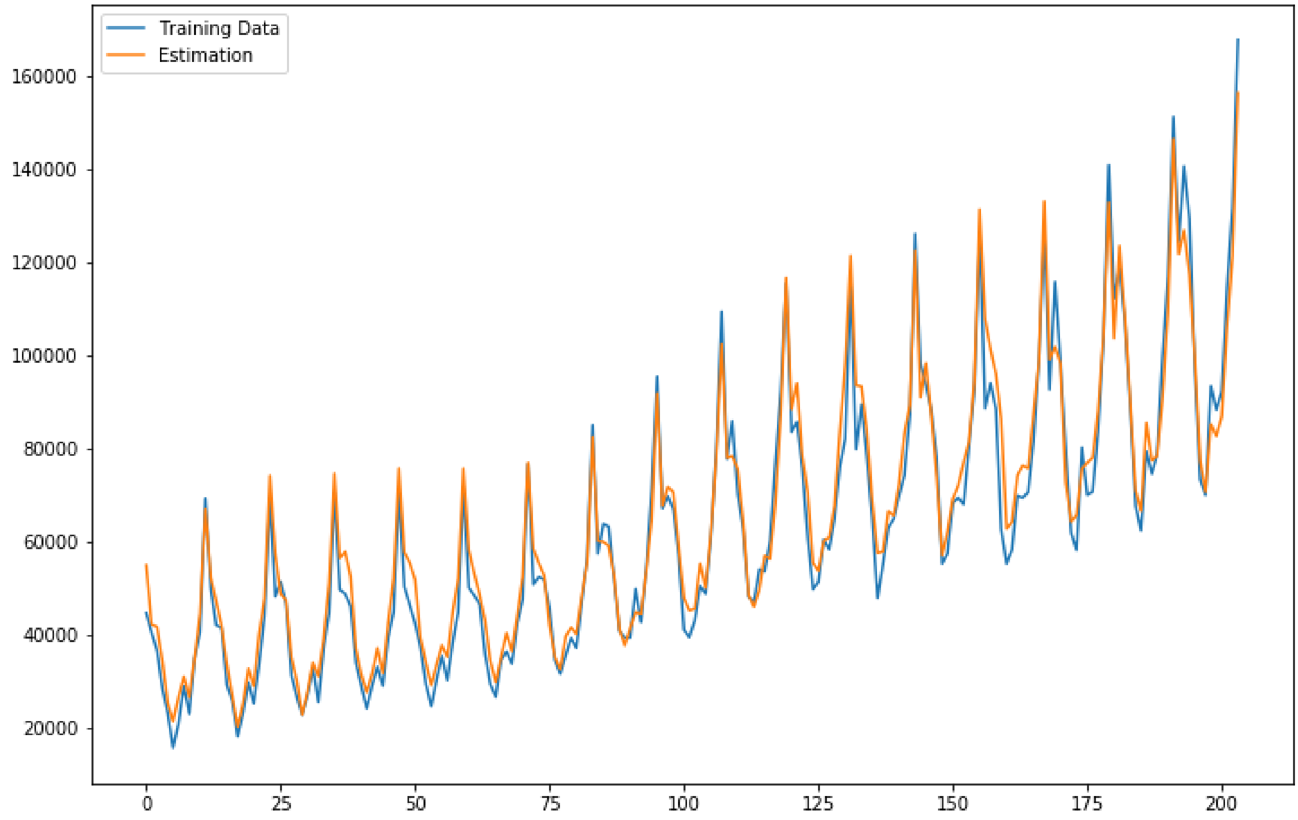
## 14. Plot in the same graph the real and estimation training data for statistical and machine learning model. What can you notice?

In [119]:
```
train_predictions =model2.predict_generator(generator)
```

In [120]:
```
true_train_predictions = scaler.inverse_transform(train_predictions)
```

In [121]:
```
original_train = scaler.inverse_transform(generator.data)
```

In [122]:
```
plt.figure(figsize=(12,8))

plt.plot(range(0,len(true_train_predictions)),original_train[-len(true_train_predicti
ons):],label="Training Data")
plt.plot(range(0,len(true_train_predictions)),true_train_predictions,label="Estimatio
n ")
plt.legend()
plt.show()
```



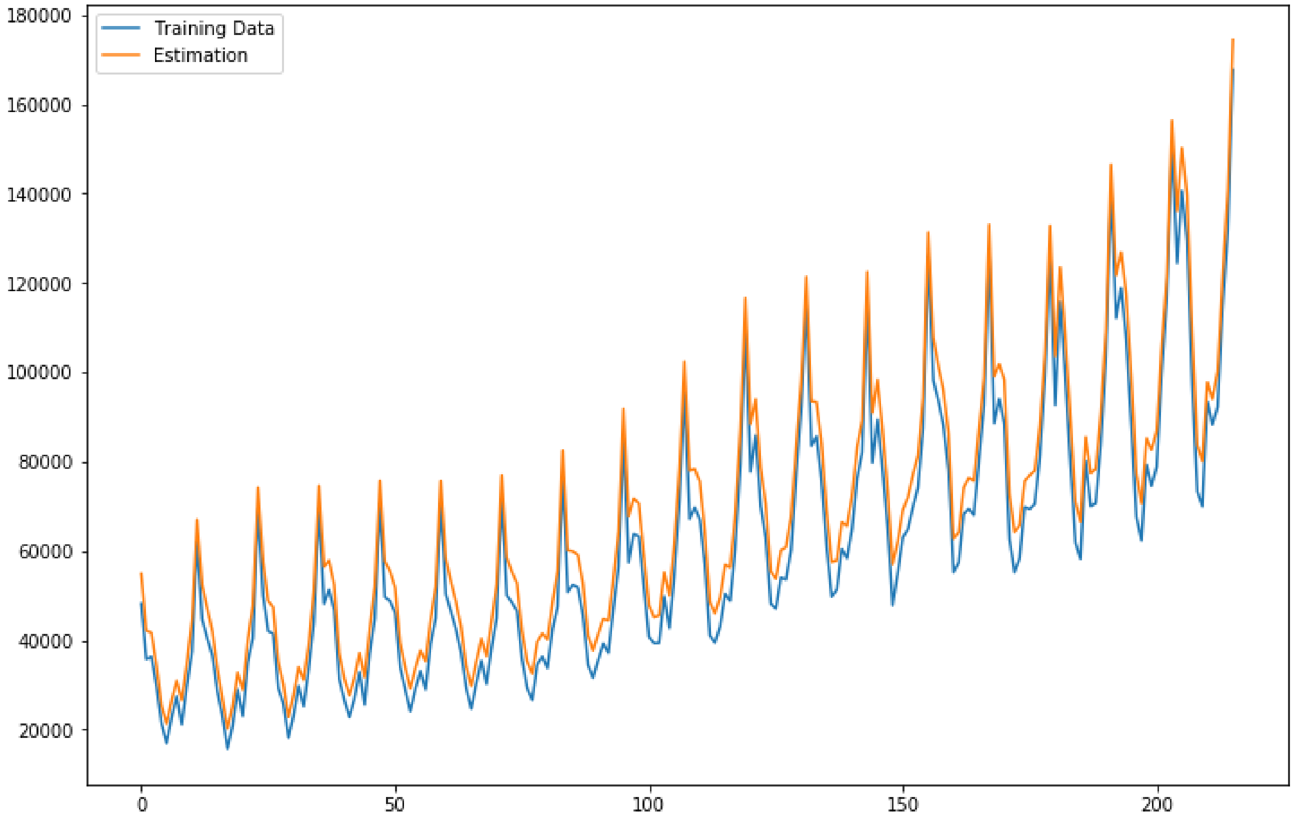

### The model has well modeled the training data and understood the logic of the data

### 15. Plot in the same graph the real and estimation testing data for statistical and machine learning model. What can you notice?

In [123]:
```
reshaped = test["Predictions"].values.reshape((len(test["Predictions"]),1))
```

In [124]:
```
all_predictions = np.concatenate((true_train_predictions,reshaped))
```

In [125]:
```
plt.figure(figsize=(12,8))

plt.plot(range(0,len(all_predictions)),df[:len(all_predictions)],label="Training Dat
a")
plt.plot(range(0,len(all_predictions)),all_predictions,label="Estimation ")
plt.legend()
plt.show()
```




## The predictions fit well the test data which means our model is good

## 16. Compare the computed training and testing performances of the statistical model and machine learning model.

**Both the machine learning model & the statistical model give good results**

## 17. What can you conclude about your data? And what are your suggestions?

**The seasonal component of the data made it harder for some models like the Arima ones to well predict its behavior, luckly the presence of models like the Sarima ones that takes into accounts the seasonality of the time series gives better results and better modelling, when it comes to machine learning the LSTM gives far better result and it doesnt require any conditions on the stationarity or the seasonality of the time series which makes this approch a solid one**

In [17]:
```
import numpy as np
import tensorflow as tf
from tensorflow import keras
import pandas as pd
import seaborn as sns
from pylab import rcParams
import matplotlib.pyplot as plt
from matplotlib import rc
from sklearn.model_selection import train_test_split
from pandas.plotting import register_matplotlib_converters
import warnings
warnings.filterwarnings('ignore')
```

# Multivariate time series

# II. Machine learning modeling:

**Metadata:**

**"timestamp" - timestamp field for grouping the data**

**"cnt" - the count of a new bike shares**

**"t1" - real temperature in C**

**"t2" - temperature in C "feels like"**

**"hum" - humidity in percentage**

**"windspeed" - wind speed in km/h**

**"weathercode" - category of the weather**

**"isholiday" - boolean field - 1 holiday / 0 non holiday**

**"isweekend" - boolean field - 1 if the day is weekend**

**"season" - category field meteorological seasons: 0-spring ; 1-summer; 2-fall; 3-winter.**

**"weathe_code" category description:**

**1 = Clear ; mostly clear but have some values with haze/fog/patches of fog/ fog in vicinity 2 = scattered clouds / few clouds 3 = Broken clouds 4 = Cloudy 7 = Rain/ light Rain shower/ Light rain 10 = rain with thunderstorm 26 = snowfall 94 = Freezing Fog**

In [4]:
```
df = pd.read_csv(
  "C:\\Users\\RABBAI\\Downloads\\london_merged.csv",
  parse_dates=['timestamp'],
  index_col="timestamp"
)
```

In [5]:
```
df.shape
```
Out[5]:
```
(17414, 9)
```

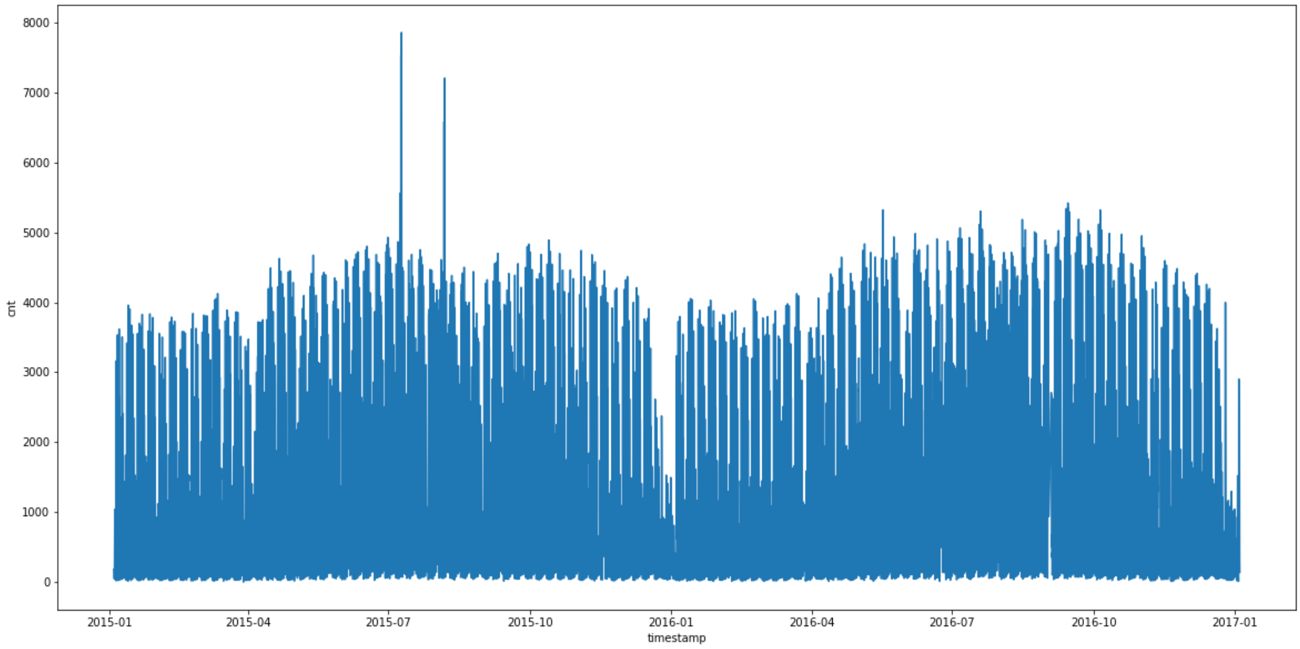
In [6]:
```
df.head()
```
Out[6]:

| | cnt | t1 | t2 | hum | wind_speed | weather_code | is_holiday | is_weekend | season |
|---|---|---|---|---|---|---|---|---|---|
| **timestamp** | | | | | | | | | |
| **2015-01-04 00:00:00** | 182 | 3.0 | 2.0 | 93.0 | 6.0 | 3.0 | 0.0 | 1.0 | 3.0 |
| **2015-01-04 01:00:00** | 138 | 3.0 | 2.5 | 93.0 | 5.0 | 1.0 | 0.0 | 1.0 | 3.0 |
| **2015-01-04 02:00:00** | 134 | 2.5 | 2.5 | 96.5 | 0.0 | 1.0 | 0.0 | 1.0 | 3.0 |
| **2015-01-04 03:00:00** | 72 | 2.0 | 2.0 | 100.0 | 0.0 | 1.0 | 0.0 | 1.0 | 3.0 |
| **2015-01-04 04:00:00** | 47 | 2.0 | 0.0 | 93.0 | 6.5 | 1.0 | 0.0 | 1.0 | 3.0 |

## Data exploration & feature selection

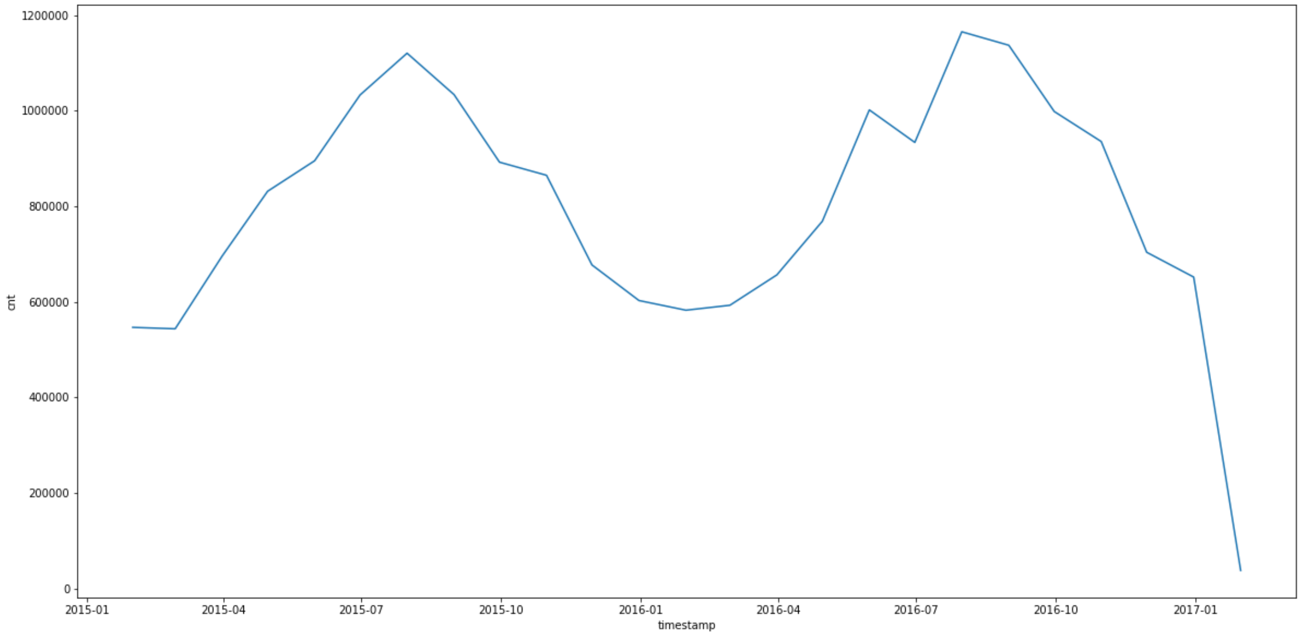
In [7]:
```
df['hour'] = df.index.hour
df['day_of_month'] = df.index.day
df['day_of_week'] = df.index.dayofweek
df['month'] = df.index.month
```

In [9]:
```
plt.figure(figsize=(20,10))
sns.lineplot(x=df.index, y="cnt", data=df);
```




## We can see that the time series has a seasonality

In [12]:
```
plt.figure(figsize=(20,10))
df_by_month = df.resample('M').sum()
sns.lineplot(x=df_by_month.index, y="cnt", data=df_by_month);
```



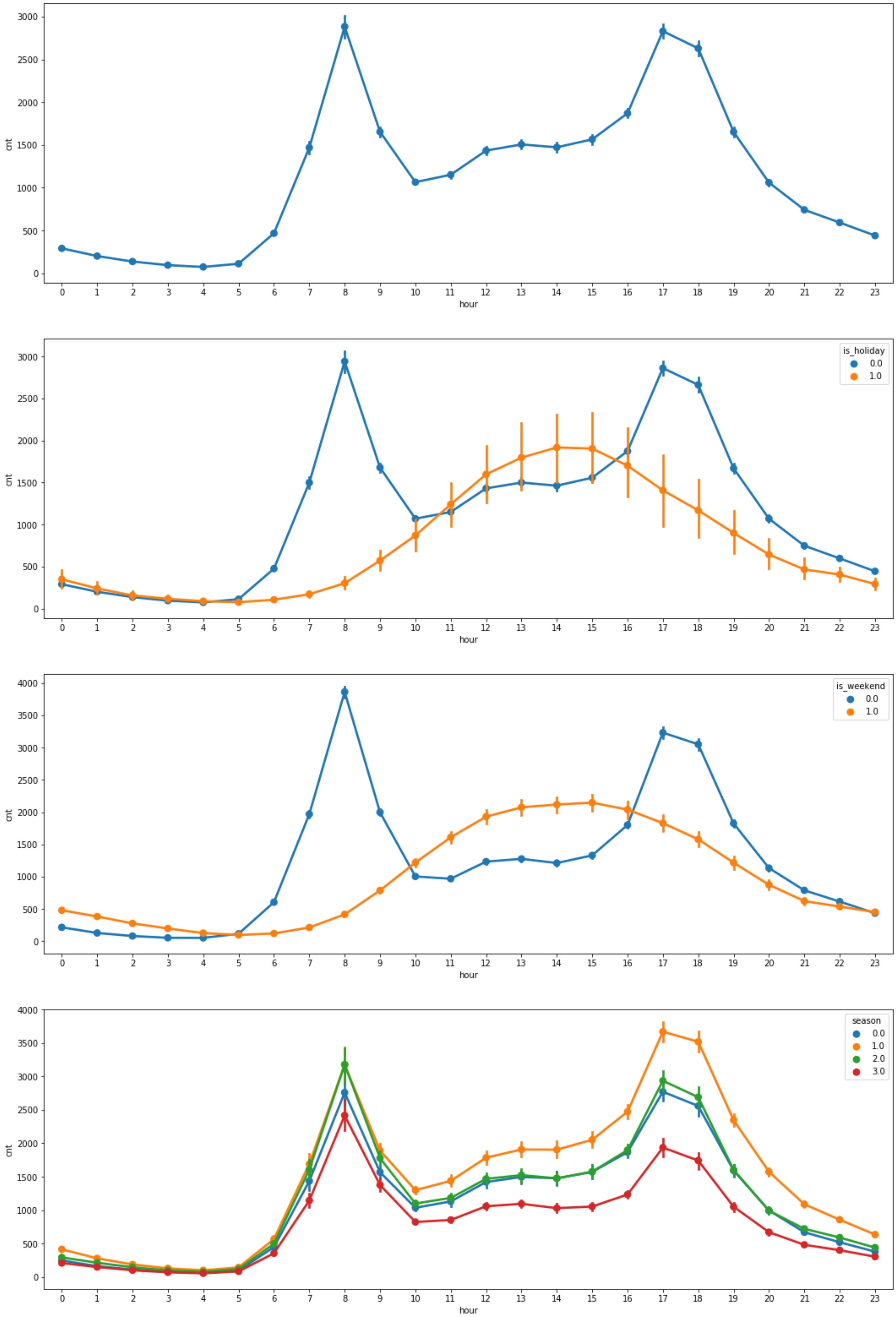

**The hour of the day, holidays, weekends, and season all affect the number of bike shares which makes them good features for our model**

**As people tend to use more bikes from 7h-10h and also at the evening from 16h-20h**

**The spikes for a holiday differ from the ones of regular days as in holidays bike shares are higher around 11h-17h**

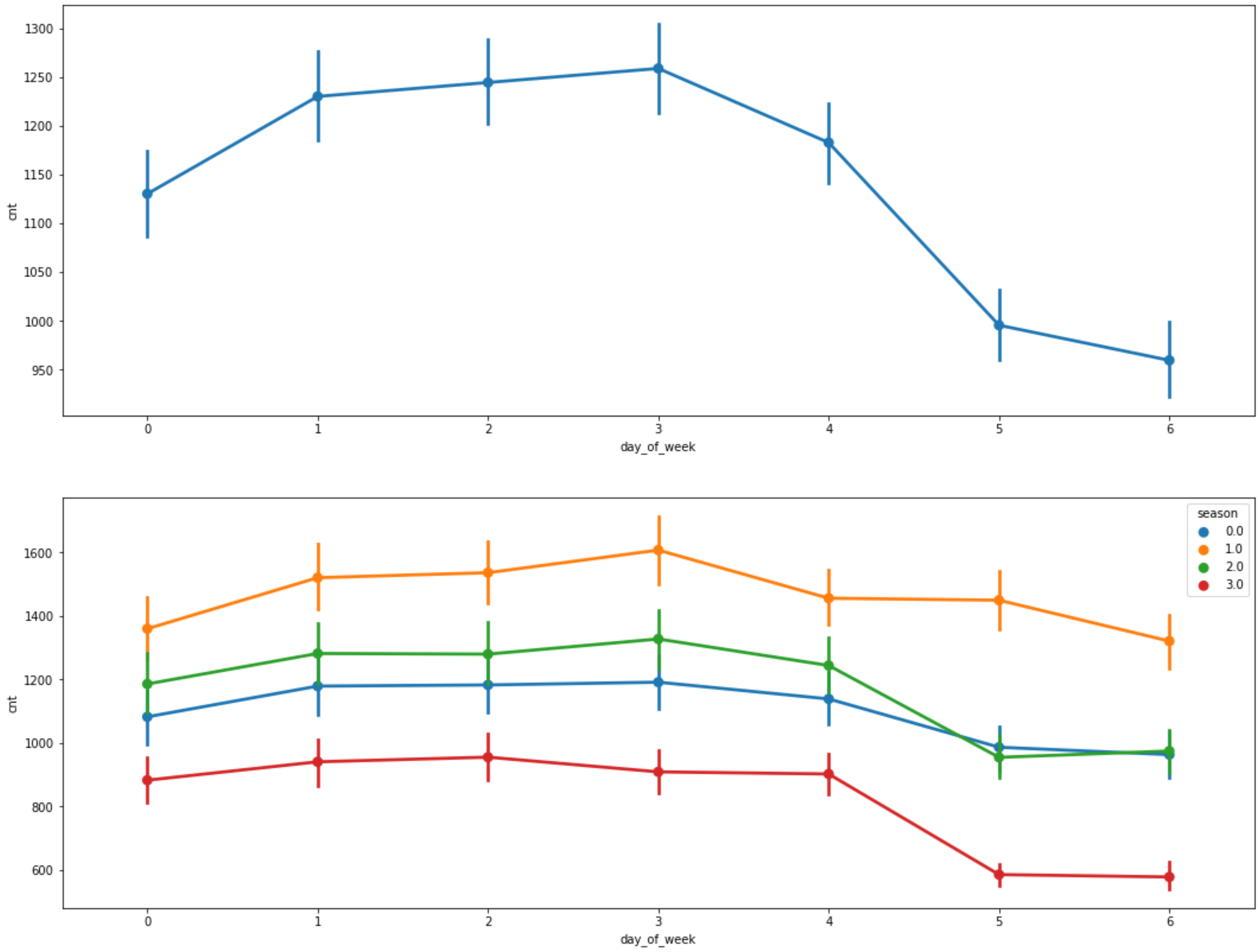
**Season also affect the general behavior as in summer the bike shares are higher & in winter they are low**

In [13]:
```python
fig,(ax1, ax2, ax3, ax4)= plt.subplots(nrows=4)
fig.set_size_inches(18, 28)
sns.pointplot(data=df, x='hour', y='cnt', ax=ax1)
sns.pointplot(data=df, x='hour', y='cnt', hue='is_holiday', ax=ax2)
sns.pointplot(data=df, x='hour', y='cnt', hue='is_weekend', ax=ax3)
sns.pointplot(data=df, x='hour', y='cnt', hue='season', ax=ax4);
```



### The days of the week also has an affect on bike shares are they are higher from monday to friday and they are lower for saturday and sunday

In [14]:
```
fig,(ax1, ax2)= plt.subplots(nrows=2)
fig.set_size_inches(18, 14)

sns.pointplot(data=df, x='day_of_week', y='cnt', ax=ax1)
sns.pointplot(data=df, x='day_of_week', y='cnt', hue='season', ax=ax2);
```




In [15]:
```
train_size = int(len(df) * 0.9)
test_size = len(df) - train_size
train, test = df.iloc[0:train_size], df.iloc[train_size:len(df)]
print(len(train), len(test))
```

```
15672 1742
```

### Data scaling

In [19]:
```
from sklearn.preprocessing import RobustScaler

f_columns = ['t1', 't2', 'hum', 'wind_speed']

f_transformer = RobustScaler()
cnt_transformer = RobustScaler()

f_transformer = f_transformer.fit(train[f_columns].to_numpy())
cnt_transformer = cnt_transformer.fit(train[['cnt']])

train.loc[:, f_columns] = f_transformer.transform(train[f_columns].to_numpy())
train['cnt'] = cnt_transformer.transform(train[['cnt']])

test.loc[:, f_columns] = f_transformer.transform(test[f_columns].to_numpy())
test['cnt'] = cnt_transformer.transform(test[['cnt']])
```

# 1. Select one machine learning model from your choice to model the given time series.

**The model that will be used to model the data is RNN (LSTM)**

# 2. Justify the choice of your model. And prove that this model can be used for time series forecasting.

**Long Short-Term Memory (LSTM) is a type of recurrent neural network that can learn the order dependence between items in a sequence. LSTMs have the promise of being able to learn the context required to make predictions in time series forecasting problems, rather than having this context pre-specified and fixed**

# 3. Provide the form of the hypothesis set of your model.

# 4. Define the parameters and the hyper-parameters of your model.

In [20]:
```
def create_dataset(X, y, time_steps=1):
    Xs, ys = [], []
    for i in range(len(X) - time_steps):
        v = X.iloc[i:(i + time_steps)].values
        Xs.append(v)
        ys.append(y.iloc[i + time_steps])
    return np.array(Xs), np.array(ys)
```

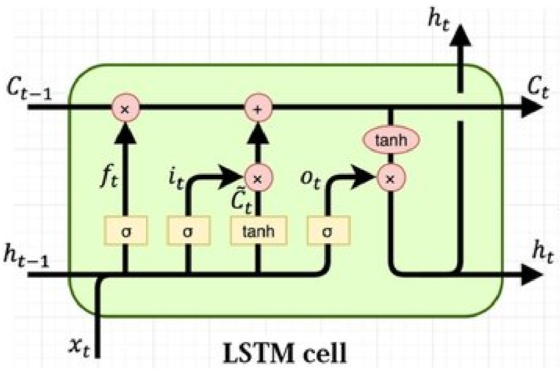
In [22]:
```
time_steps = 24

# reshape to [samples, time_steps, n_features]

X_train, y_train = create_dataset(train, train.cnt, time_steps)
X_test, y_test = create_dataset(test, test.cnt, time_steps)

print(X_train.shape, y_train.shape)
```

```
(15648, 24, 13) (15648,)
```

### Creating the model

In [23]:
```
model = keras.Sequential()
model.add(
  keras.layers.Bidirectional(
    keras.layers.LSTM(
      units=128,
      input_shape=(X_train.shape[1], X_train.shape[2])
    )
  )
)
model.add(keras.layers.Dropout(rate=0.2))
model.add(keras.layers.Dense(units=1))
model.compile(loss='mean_squared_error', optimizer='adam')
```

## 5. Explain the functioning of the selected model using mathematical equations and graphs.




$$i_t = \sigma\left(x_t U^i + h_{t-1} W^i\right)$$
$$f_t = \sigma\left(x_t U^f + h_{t-1} W^f\right)$$
$$o_t = \sigma\left(x_t U^o + h_{t-1} W^o\right)$$
$$\tilde{C}_t = \tanh\left(x_t U^g + h_{t-1} W^g\right)$$
$$C_t = \sigma\left(f_t * C_{t-1} + i_t * \tilde{C}_t\right)$$
$$h_t = \tanh(C_t) * o_t$$

## 6. Provide a set of candidate models. Justify your choice.

### Without Dropout

In [25]:
```
model2 = keras.Sequential()
model2.add(
  keras.layers.Bidirectional(
    keras.layers.LSTM(
      units=128,
      input_shape=(X_train.shape[1], X_train.shape[2])
    )
  )
)
model2.add(keras.layers.Dense(units=1))
model2.compile(loss='mean_squared_error', optimizer='adam')
```

### With Dropout

```
In [ ]: model = keras.Sequential()
        model.add(
          keras.layers.Bidirectional(
            keras.layers.LSTM(
              units=128,
              input_shape=(X_train.shape[1], X_train.shape[2])
            )
          )
        )
        model.add(keras.layers.Dropout(rate=0.2))
        model.add(keras.layers.Dense(units=1))
        model.compile(loss='mean_squared_error', optimizer='adam')
```

## 7. Choose one learning algorithm to minimize your cost function. Justify your choice.

***a) Select a learning rate of your algorithm (constant or adaptive learning rate). Justify your choice. If it is adaptive provide the adaption technique.***

***The learning rate is adaptive, since this one provides better results & adapts to the data, the optimizer used is ADAM (adaptive moment estimation optimizer)***

## 8. Use your models to learn the parameters based on training data, and save the training performances.

In [24]:
```
history = model.fit(
    X_train, y_train,
    epochs=30,
    batch_size=32,
    validation_split=0.1,
    shuffle=False
)
```

```
Epoch 1/30
441/441 [==============================] - 23s 43ms/step - loss: 0.3625 - val_loss:
0.1745
Epoch 2/30
441/441 [==============================] - 16s 37ms/step - loss: 0.1220 - val_loss:
0.0863
Epoch 3/30
441/441 [==============================] - 16s 36ms/step - loss: 0.0724 - val_loss:
0.0455
Epoch 4/30
441/441 [==============================] - 16s 37ms/step - loss: 0.0544 - val_loss:
0.0375
Epoch 5/30
441/441 [==============================] - 16s 36ms/step - loss: 0.0374 - val_loss:
0.0468
Epoch 6/30
441/441 [==============================] - 16s 36ms/step - loss: 0.0315 - val_loss:
0.0393
Epoch 7/30
441/441 [==============================] - 16s 37ms/step - loss: 0.0285 - val_loss:
0.0335
Epoch 8/30
441/441 [==============================] - 17s 38ms/step - loss: 0.0252 - val_loss:
0.0362
Epoch 9/30
441/441 [==============================] - 16s 37ms/step - loss: 0.0242 - val_loss:
0.0310
Epoch 10/30
441/441 [==============================] - 16s 37ms/step - loss: 0.0219 - val_loss:
0.0414
Epoch 11/30
441/441 [==============================] - 16s 36ms/step - loss: 0.0212 - val_loss:
0.0353
Epoch 12/30
441/441 [==============================] - 16s 35ms/step - loss: 0.0204 - val_loss:
0.0326
Epoch 13/30
441/441 [==============================] - 16s 36ms/step - loss: 0.0193 - val_loss:
0.0328
Epoch 14/30
441/441 [==============================] - 16s 37ms/step - loss: 0.0175 - val_loss:
0.0279
Epoch 15/30
441/441 [==============================] - 17s 37ms/step - loss: 0.0175 - val_loss:
0.0274
Epoch 16/30
441/441 [==============================] - 17s 39ms/step - loss: 0.0178 - val_loss:
0.0335
Epoch 17/30
441/441 [==============================] - 16s 37ms/step - loss: 0.0169 - val_loss:
0.0272
Epoch 18/30
441/441 [==============================] - 17s 38ms/step - loss: 0.0154 - val_loss:
0.0304
Epoch 19/30
441/441 [==============================] - 18s 40ms/step - loss: 0.0162 - val_loss:
0.0228
Epoch 20/30
441/441 [==============================] - 17s 38ms/step - loss: 0.0154 - val_loss:
0.0249
Epoch 21/30
441/441 [==============================] - 17s 38ms/step - loss: 0.0148 - val_loss:
0.0261
Epoch 22/30
441/441 [==============================] - 17s 37ms/step - loss: 0.0151 - val_loss:
```

```
0.0310
Epoch 23/30
441/441 [==============================] - 17s 39ms/step - loss: 0.0144 - val_loss:
0.0221
Epoch 24/30
441/441 [==============================] - 16s 37ms/step - loss: 0.0149 - val_loss:
0.0277
Epoch 25/30
441/441 [==============================] - 17s 37ms/step - loss: 0.0139 - val_loss:
0.0275
Epoch 26/30
441/441 [==============================] - 17s 37ms/step - loss: 0.0141 - val_loss:
0.0296
Epoch 27/30
441/441 [==============================] - 17s 38ms/step - loss: 0.0127 - val_loss:
0.0268
Epoch 28/30
441/441 [==============================] - 16s 36ms/step - loss: 0.0129 - val_loss:
0.0250
Epoch 29/30
441/441 [==============================] - 16s 36ms/step - loss: 0.0131 - val_loss:
0.0323
Epoch 30/30
441/441 [==============================] - 17s 38ms/step - loss: 0.0131 - val_loss:
0.0241
```

In [26]:
```
history = model2.fit(
    X_train, y_train,
    epochs=30,
    batch_size=32,
    validation_split=0.1,
    shuffle=False
)
```

```
Epoch 1/30
441/441 [==============================] - 20s 38ms/step - loss: 0.3601 - val_loss:
0.1591
Epoch 2/30
441/441 [==============================] - 16s 36ms/step - loss: 0.1123 - val_loss:
0.0771
Epoch 3/30
441/441 [==============================] - 16s 37ms/step - loss: 0.0636 - val_loss:
0.0681
Epoch 4/30
441/441 [==============================] - 16s 36ms/step - loss: 0.0362 - val_loss:
0.0445
Epoch 5/30
441/441 [==============================] - 16s 37ms/step - loss: 0.0316 - val_loss:
0.0415
Epoch 6/30
441/441 [==============================] - 16s 36ms/step - loss: 0.0316 - val_loss:
0.0354
Epoch 7/30
441/441 [==============================] - 16s 37ms/step - loss: 0.0270 - val_loss:
0.0357
Epoch 8/30
441/441 [==============================] - 16s 37ms/step - loss: 0.0262 - val_loss:
0.0329
Epoch 9/30
441/441 [==============================] - 16s 36ms/step - loss: 0.0253 - val_loss:
0.0304
Epoch 10/30
441/441 [==============================] - 16s 36ms/step - loss: 0.0243 - val_loss:
0.0285
Epoch 11/30
441/441 [==============================] - 16s 37ms/step - loss: 0.0239 - val_loss:
0.0300
Epoch 12/30
441/441 [==============================] - 16s 36ms/step - loss: 0.0237 - val_loss:
0.0319
Epoch 13/30
441/441 [==============================] - 16s 37ms/step - loss: 0.0216 - val_loss:
0.0312
Epoch 14/30
441/441 [==============================] - 16s 36ms/step - loss: 0.0186 - val_loss:
0.0312
Epoch 15/30
441/441 [==============================] - 16s 37ms/step - loss: 0.0180 - val_loss:
0.0304
Epoch 16/30
441/441 [==============================] - 16s 36ms/step - loss: 0.0160 - val_loss:
0.0286
Epoch 17/30
441/441 [==============================] - 17s 38ms/step - loss: 0.0138 - val_loss:
0.0282
Epoch 18/30
441/441 [==============================] - 16s 36ms/step - loss: 0.0131 - val_loss:
0.0299
Epoch 19/30
441/441 [==============================] - 16s 37ms/step - loss: 0.0127 - val_loss:
0.0267
Epoch 20/30
441/441 [==============================] - 18s 41ms/step - loss: 0.0122 - val_loss:
0.0348
Epoch 21/30
441/441 [==============================] - 16s 36ms/step - loss: 0.0126 - val_loss:
0.0286
Epoch 22/30
441/441 [==============================] - 16s 37ms/step - loss: 0.0114 - val_loss:
```

```
0.0296
Epoch 23/30
441/441 [==============================] - 16s 35ms/step - loss: 0.0111 - val_loss:
0.0315
Epoch 24/30
441/441 [==============================] - 16s 35ms/step - loss: 0.0107 - val_loss:
0.0370
Epoch 25/30
441/441 [==============================] - 15s 35ms/step - loss: 0.0105 - val_loss:
0.0346
Epoch 26/30
441/441 [==============================] - 16s 35ms/step - loss: 0.0108 - val_loss:
0.0354
Epoch 27/30
441/441 [==============================] - 16s 36ms/step - loss: 0.0102 - val_loss:
0.0369
Epoch 28/30
441/441 [==============================] - 16s 35ms/step - loss: 0.0096 - val_loss:
0.0346
Epoch 29/30
441/441 [==============================] - 16s 35ms/step - loss: 0.0093 - val_loss:
0.0294
Epoch 30/30
441/441 [==============================] - 16s 37ms/step - loss: 0.0104 - val_loss:
0.0329
```

### 9. Select the best model using the validation data.

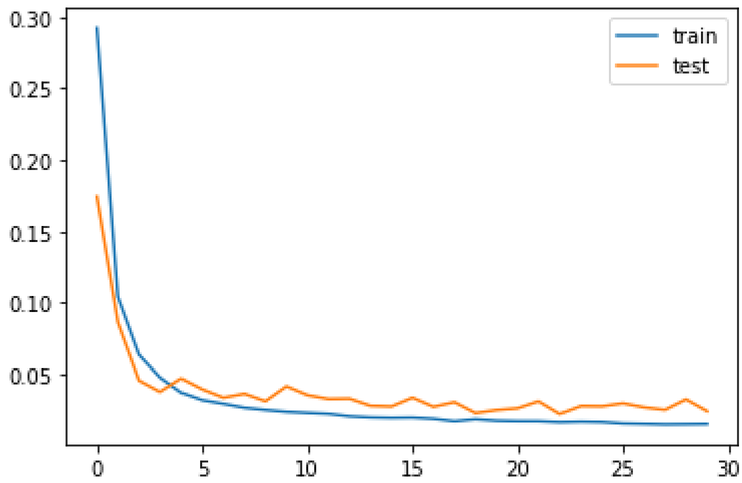
In [30]:
```python
print("Model 1 Val loss = ", model.history.history["val_loss"][-1])
print("Model 2 Val loss = ", model2.history.history["val_loss"][-1])
```

```
Model 1 Val loss =  0.02411140501499176
Model 2 Val loss =  0.03290883079171181
```

### We'll continue using Model 1

### 10. Test your selected model on the testing data and retain your results based on the statistical metrics.

In [28]:
```python
plt.plot(model.history.history['loss'], label='train')
plt.plot(model.history.history['val_loss'], label='test')
plt.legend();
```



## 11. Is there any overfitting or underfitting problem?

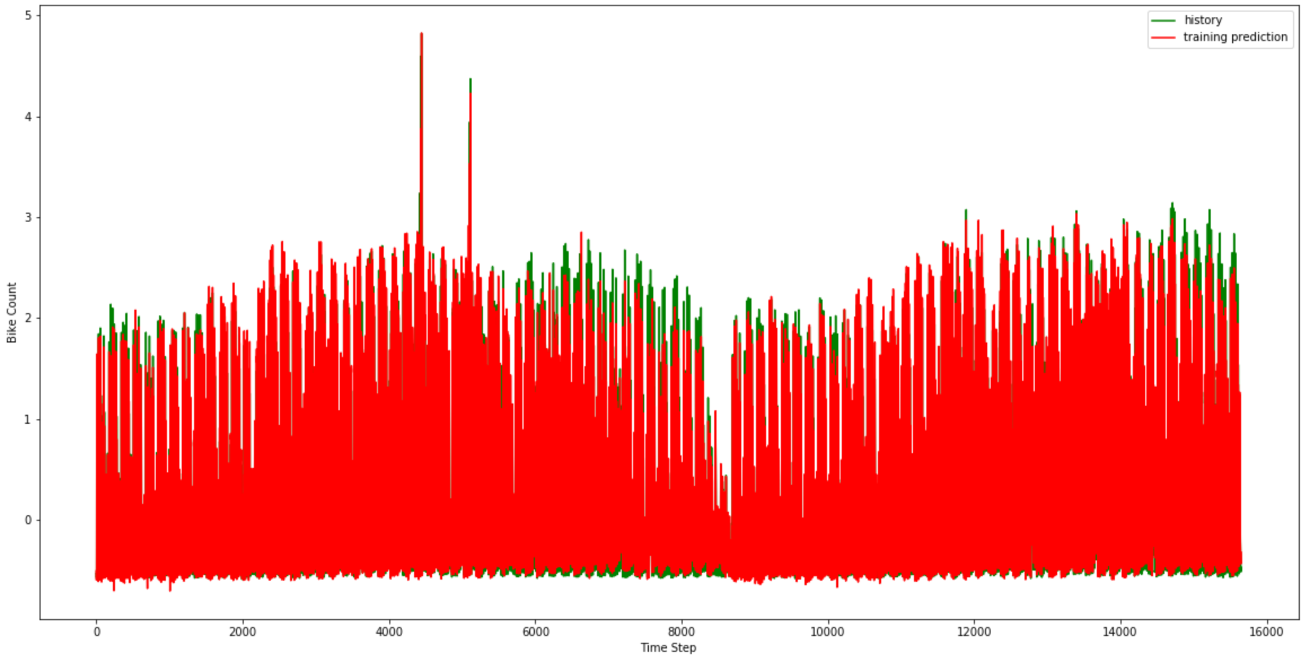
**The validation loss dont increase at the end which indicates the absence of overfitting**

## 14. Plot in the same graph the real and estimation training data for statistical and machine learning model. What can you notice?

In [42]:
```
y_train_pred = model.predict(X_train)
y_pred = model.predict(X_test)
```

In [43]:
```
y_train_inv = cnt_transformer.inverse_transform(y_train.reshape(1, -1))
y_test_inv = cnt_transformer.inverse_transform(y_test.reshape(1, -1))
y_pred_inv = cnt_transformer.inverse_transform(y_pred)
y_train_pred_inv = cnt_transformer.inverse_transform(y_train_pred)
```

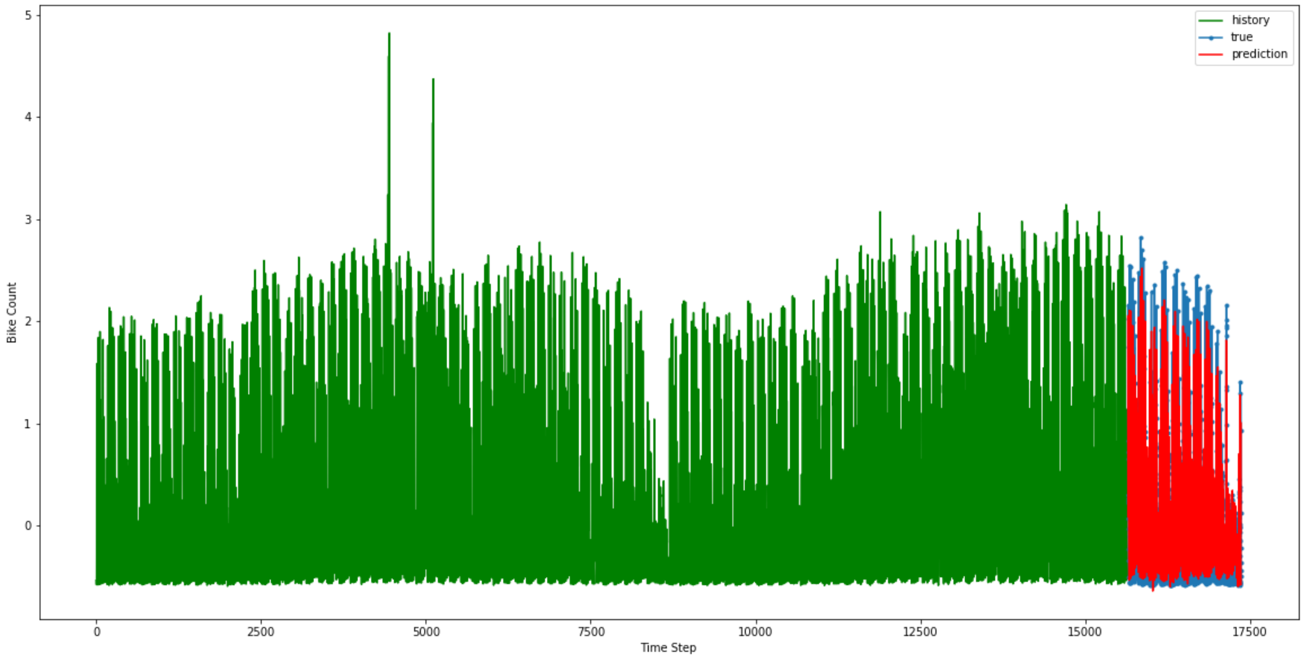
In [46]:
```
plt.figure(figsize=(20,10))
plt.plot(np.arange(0, len(y_train)), y_train_inv.flatten(), 'g', label="history")

plt.plot(np.arange(0, len(y_train)), y_train_pred_inv.flatten(), 'r', label="training prediction")
plt.ylabel('Bike Count')
plt.xlabel('Time Step')
plt.legend()
plt.show();
```




**The model has manage to well capture the data even some outliers**

## 15. Plot in the same graph the real and estimation testing data for statistical and machine learning model. What can you notice?

In [47]:
```python
plt.figure(figsize=(20,10))
plt.plot(np.arange(0, len(y_train)), y_train_inv.flatten(), 'g', label="history")
plt.plot(np.arange(len(y_train), len(y_train) + len(y_test)), y_test_inv.flatten(), m
arker='.', label="true")
plt.plot(np.arange(len(y_train), len(y_train) + len(y_test)), y_pred_inv.flatten(),
'r', label="prediction")
plt.ylabel('Bike Count')
plt.xlabel('Time Step')
plt.legend()
plt.show();
```




**The predictions fit well the test data which means our model is good**