In [194]:
```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_c
sv)
import json
import os
import pandas as pd
import seaborn as sns
#import statsmodels.formula.api as smf
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import LabelEncoder
from sklearn import metrics
from sklearn.model_selection  import train_test_split
import numpy as np
import gc
from scipy.stats import norm # for scientific Computing
from scipy import stats, integrate
import matplotlib.pyplot as plt
from sklearn import preprocessing
from keras import backend as K
from keras.callbacks import ModelCheckpoint, EarlyStopping
from keras.layers import Dense, LSTM, GRU, Dropout, BatchNormalizat
ion
from keras.models import Sequential
from keras.optimizers import RMSprop,Adam
from keras import regularizers
from keras import layers
from time import time
## Suppress warnings
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
from tensorflow.python.util import deprecation
deprecation._PRINT_DEPRECATION_WARNINGS = False
```

Supporting Functions

In [195]:
```
def reduce_memory_usage(df, verbose=True):
    numerics = ['int16', 'int32', 'int64', 'float16', 'float32', 'f
loat64']
    start_mem = df.memory_usage().sum() / 1024**2
    for col in df.columns:
        col_type = df[col].dtypes
        if col_type in numerics:
            c_min = df[col].min()
            c_max = df[col].max()
            if str(col_type)[:3] == 'int':
                if c_min > np.iinfo(np.int8).min and c_max < np.iin
fo(np.int8).max:
                    df[col] = df[col].astype(np.int8)
                elif c_min > np.iinfo(np.int16).min and c_max < np.
iinfo(np.int16).max:
                    df[col] = df[col].astype(np.int16)
                elif c_min > np.iinfo(np.int32).min and c_max < np.
iinfo(np.int32).max:
                    df[col] = df[col].astype(np.int32)
                elif c_min > np.iinfo(np.int64).min and c_max < np.
iinfo(np.int64).max:
                    df[col] = df[col].astype(np.int64)
            else:
                if c_min > np.finfo(np.float16).min and c_max < np.
finfo(np.float16).max:
                    df[col] = df[col].astype(np.float16)
                elif c_min > np.finfo(np.float32).min and c_max < n
p.finfo(np.float32).max:
                    df[col] = df[col].astype(np.float32)
                else:
                    df[col] = df[col].astype(np.float64)
    end_mem = df.memory_usage().sum() / 1024**2
    if verbose: print('Mem. usage decreased to {:5.2f} Mb ({:.1f}%
reduction)'.format(end_mem, 100 * (start_mem - end_mem) / start_me
m))
    return df

def generator(data, lookback, delay, min_index, max_index,
              shuffle=False, batch_size=128, step=6):
    if max_index is None:
        max_index = len(data) - delay - 1
    i = min_index + lookback
    while 1:
        if shuffle:
            rows = np.random.randint(
                min_index + lookback, max_index, size=batch_size)
        else:
            if i + batch_size >= max_index:
                i = min_index + lookback
            rows = np.arange(i, min(i + batch_size, max_index))
            i += len(rows)

        samples = np.zeros((len(rows),
                            lookback // step,
```

Load data

In [196]:
```
ASHRAE_train = pd.read_csv('train.csv')
ASHRAE_test = pd.read_csv('test.csv')
weather_train = pd.read_csv('weather_train.csv')
weather_test = pd.read_csv('weather_test.csv')
building_meta = pd.read_csv('building_metadata.csv')
meter_readings = pd.read_csv('meter_readings.csv')

## Reduce Memory
reduce_memory_usage(building_meta)
reduce_memory_usage(weather_train)
reduce_memory_usage(ASHRAE_train)
reduce_memory_usage(weather_test)
reduce_memory_usage(ASHRAE_test)
reduce_memory_usage(meter_readings)
```

```
Mem. usage decreased to  0.03 Mb (60.3% reduction)
Mem. usage decreased to  3.07 Mb (68.1% reduction)
Mem. usage decreased to 289.19 Mb (53.1% reduction)
Mem. usage decreased to  6.08 Mb (68.1% reduction)
Mem. usage decreased to 596.49 Mb (53.1% reduction)
Mem. usage decreased to 318.13 Mb (50.0% reduction)
```

Out[196]:

	row_id	meter_reading
0	0	195.209305
1	1	89.766899
2	2	8.549000
3	3	304.708313
4	4	1213.726440
5	5	17.076700
6	6	109.895599
7	7	468.770599
8	8	887.263916
9	9	383.855804
10	10	64.180603
11	11	14.352200
12	12	1077.743774
13	13	385.463104
14	14	231.925400
15	15	215.721603
16	16	82.893898
17	17	292.329407
18	18	624.532715
19	19	190.222504
20	20	474.945892
21	21	1097.555176
22	22	105.756302
23	23	1840.520142
24	24	175.480896
25	25	379.871094
26	26	51.933399
27	27	21.147499
28	28	614.862488
29	29	645.872192
...	...	...
41697570	41697570	1736.225342
41697571	41697571	47.108398

Merge Data

In [197]:
```
BuildingTrain = building_meta.merge(ASHRAE_train, left_on='building
_id', right_on='building_id' , how='left')
MeterTest = meter_readings.merge(ASHRAE_test, left_on='row_id', rig
ht_on='row_id' , how='left')
BuildingTest = building_meta.merge(MeterTest, left_on='building_id
', right_on='building_id' , how='left')
BTW_train=BuildingTrain.merge(weather_train,left_on=['site_id','tim
estamp'],right_on=['site_id','timestamp'],how='left')
BTW_test = BuildingTest.merge(weather_test,left_on=['site_id','time
stamp'],right_on=['site_id','timestamp'],how='left')
BTW_train.shape
```

Out[197]: (20216100, 16)

In [198]:
```
BTW_test["meter_reading"].mean()
```

Out[198]: 435.2048034667969

Explore Data

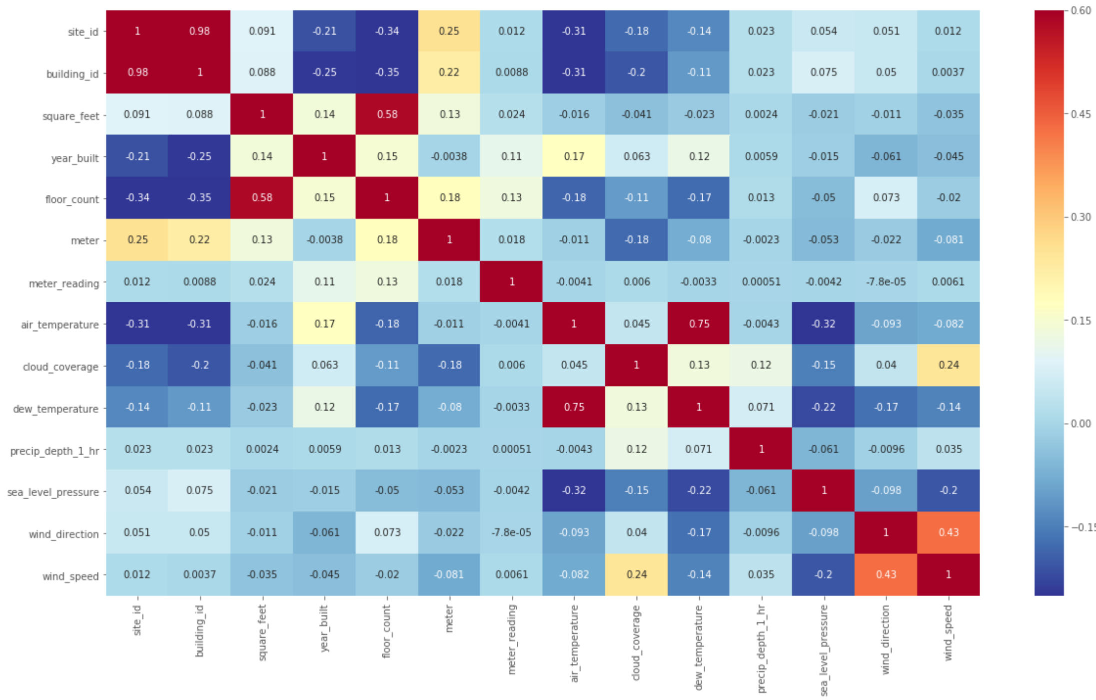
In [199]:
```
BTW_test.describe()
```

Out[199]:

	site_id	building_id	square_feet	year_built	floor_count	row_id
count	4.169760e+07	4.169760e+07	4.169760e+07	17099520.0	7253280.0	4.169760e+07
mean	8.086134e+00	8.075824e+02	1.069469e+05	NaN	NaN	2.084880e+07
std	5.134712e+00	4.297680e+02	1.160888e+05	NaN	0.0	1.203706e+07
min	0.000000e+00	0.000000e+00	2.830000e+02	1900.0	1.0	0.000000e+00
25%	3.000000e+00	4.047500e+02	3.224350e+04	1951.0	1.0	1.042440e+07
50%	9.000000e+00	9.000000e+02	7.226250e+04	1969.0	3.0	2.084880e+07
75%	1.300000e+01	1.194250e+03	1.383875e+05	1993.0	6.0	3.127320e+07
max	1.500000e+01	1.448000e+03	8.750000e+05	2017.0	26.0	4.169760e+07

In [200]:
```
corrmat=BTW_train.corr()
plt.figure(figsize = (20,11))
sns.heatmap(corrmat,cmap=plt.cm.RdYlBu_r,vmin=-0.25,
            annot=True,vmax=0.6)
```

Out[200]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3cabee17b8>



Data Preparation

In [201]:
```
## Remove variables that aren't correlated with meter_reading
BTW_train = BTW_train.drop(columns=['year_built', 'floor_count', 'w
ind_direction', 'dew_temperature'])
BTW_test = BTW_test.drop(columns=['year_built', 'floor_count', 'win
d_direction', 'dew_temperature'])
```

In [202]:
```
BTW_train['timestamp'] =  pd.to_datetime(BTW_train['timestamp'])
BTW_test['timestamp'] =  pd.to_datetime(BTW_test['timestamp'])
BTW_train['Year']=pd.DatetimeIndex(BTW_train['timestamp']).year
BTW_test['Year']=pd.DatetimeIndex(BTW_test['timestamp']).year
BTW_train['Month']=pd.DatetimeIndex(BTW_train['timestamp']).month
BTW_test['Month']=pd.DatetimeIndex(BTW_test['timestamp']).month
BTW_train['Day']=pd.DatetimeIndex(BTW_train['timestamp']).day
BTW_test['Day']=pd.DatetimeIndex(BTW_test['timestamp']).day
BTW_train['Hour']=pd.DatetimeIndex(BTW_train['timestamp']).hour
BTW_test['Hour']=pd.DatetimeIndex(BTW_test['timestamp']).hour
BTW_train['QuarterDay'] = (BTW_train['Hour']-1)//6 + 1
BTW_train.loc[BTW_train['QuarterDay']==0] = 4
BTW_test['QuarterDay'] = (BTW_test['Hour']-1)//6 + 1
BTW_test.loc[BTW_test['QuarterDay']==0] = 4
BTW_train
```

Out[202]:

	site_id	building_id	primary_use	square_feet	meter	timestamp	meter_reading
0	4	4	4	4	4	4	4.000
1	0	0	Education	7432	0	2016-01-01 01:00:00	0.000
2	0	0	Education	7432	0	2016-01-01 02:00:00	0.000
3	0	0	Education	7432	0	2016-01-01 03:00:00	0.000
4	0	0	Education	7432	0	2016-01-01 04:00:00	0.000
5	0	0	Education	7432	0	2016-01-01 05:00:00	0.000
6	0	0	Education	7432	0	2016-01-01 06:00:00	0.000
7	0	0	Education	7432	0	2016-01-01 07:00:00	0.000
8	0	0	Education	7432	0	2016-01-01 08:00:00	0.000
9	0	0	Education	7432	0	2016-01-01 09:00:00	0.000
10	0	0	Education	7432	0	2016-01-01 10:00:00	0.000
11	0	0	Education	7432	0	2016-01-01 11:00:00	0.000
12	0	0	Education	7432	0	2016-01-01 12:00:00	0.000
13	0	0	Education	7432	0	2016-01-01 13:00:00	0.000
14	0	0	Education	7432	0	2016-01-01 14:00:00	0.000
15	0	0	Education	7432	0	2016-01-01 15:00:00	0.000
16	0	0	Education	7432	0	2016-01-01 16:00:00	0.000
17	0	0	Education	7432	0	2016-01-01 17:00:00	0.000
18	0	0	Education	7432	0	2016-01-01 18:00:00	0.000
19	0	0	Education	7432	0	2016-01-01 19:00:00	0.000
20	0	0	Education	7432	0	2016-01-01 20:00:00	0.000
21	0	0	Education	7432	0	2016-01-01	0.000

In [203]:
```
BTW_train_group= BTW_train.groupby(['meter',BTW_train['building_id
'],'primary_use',BTW_train['Year'], BTW_train['Month'], BTW_train['
Day'], BTW_train['QuarterDay']]).agg({'meter_reading':'sum', 'air_t
emperature': 'mean', 'wind_speed': 'mean', 'precip_depth_1_hr': 'me
an', 'cloud_coverage': 'mean', 'square_feet': 'mean'})
BTW_test_group= BTW_test.groupby(['meter',BTW_test['building_id'],'
primary_use',BTW_test['Year'], BTW_test['Month'], BTW_test['Day'],
BTW_train['QuarterDay']]).agg({'meter_reading':'sum', 'air_temperat
ure': 'mean', 'wind_speed': 'mean', 'precip_depth_1_hr': 'mean', 'c
loud_coverage': 'mean', 'square_feet': 'mean'})
BTW_train = BTW_train_group.reset_index()
BTW_test = BTW_test_group.reset_index()

BTW_train['wind_speed'] = BTW_train['wind_speed'].astype('float32')
BTW_train['air_temperature'] = BTW_train['air_temperature'].astype
('float32')
BTW_train['precip_depth_1_hr'] = BTW_train['precip_depth_1_hr'].ast
ype('float32')
BTW_train['cloud_coverage'] = BTW_train['cloud_coverage'].astype('f
loat32')

BTW_test['wind_speed'] = BTW_test['wind_speed'].astype('float32')
BTW_test['air_temperature'] = BTW_test['air_temperature'].astype('f
loat32')
BTW_test['precip_depth_1_hr'] = BTW_test['precip_depth_1_hr'].astyp
e('float32')
BTW_test['cloud_coverage'] = BTW_test['cloud_coverage'].astype('flo
at32')

BTW_train['Year'] = BTW_train['Year'].astype('float32')
BTW_train['Month'] = BTW_train['Month'].astype('float32')
BTW_train['Day'] = BTW_train['Day'].astype('float32')
BTW_train['QuarterDay'] = BTW_train['QuarterDay'].astype('float32')
BTW_test['Year'] = BTW_test['Year'].astype('float32')
BTW_test['Month'] = BTW_test['Month'].astype('float32')
BTW_test['Day'] = BTW_test['Day'].astype('float32')
BTW_test['QuarterDay'] = BTW_test['QuarterDay'].astype('float32')
```

In [204]:
```
## Missing Data
BTW_train['precip_depth_1_hr'].fillna(BTW_train['precip_depth_1_hr
'].mean(), inplace=True)
BTW_train['cloud_coverage'].fillna(BTW_train['cloud_coverage'].mean
(), inplace=True)
BTW_train['wind_speed'].fillna(BTW_train['wind_speed'].mean(), inpl
ace=True)
BTW_train['air_temperature'].fillna(BTW_train['air_temperature'].me
an(), inplace=True)

BTW_test['precip_depth_1_hr'].fillna(BTW_test['precip_depth_1_hr'].
mean(), inplace=True)
BTW_test['cloud_coverage'].fillna(BTW_test['cloud_coverage'].mean
(), inplace=True)
BTW_test['wind_speed'].fillna(BTW_test['wind_speed'].mean(), inplac
e=True)
BTW_test['air_temperature'].fillna(BTW_test['air_temperature'].mean
(), inplace=True)
BTW_train.isnull().sum()
```

Out[204]:
```
meter                0
building_id          0
primary_use          0
Year                 0
Month                0
Day                  0
QuarterDay           0
meter_reading        0
air_temperature      0
wind_speed           0
precip_depth_1_hr    0
cloud_coverage       0
square_feet          0
dtype: int64
```

In [205]:
```
BTW_train.describe().astype(int)
```

Out[205]:

	meter	building_id	Year	Month	Day	QuarterDay	meter_reading	air_te
count	3376344	3376344	3376344	3376344	3376344	3376344	3376344	
mean	0	799	2045	6	15	2	12137	
std	0	426	29	3	8	1	874566	
min	0	0	4	1	1	1	0	
25%	0	394	2016	4	8	2	115	
50%	0	895	2016	7	16	3	469	
75%	1	1179	2016	10	23	3	1567	
max	4	1448	2016	12	31	4	124445000	

In [206]:
```
## Label Encoding
#le = LabelEncoder()
BTW_encoded = BTW_train[:]
BTW_test_encoded = BTW_test[:]
#BTW_encoded["primary_use"] = le.fit_transform(BTW_encoded["primary_use"])
#BTW_test_encoded["primary_use"] = le.fit_transform(BTW_test_encoded["primary_use"])
```

Choose one building to focus on...

In [207]:
```
building2 = BTW_encoded[(BTW_encoded.building_id == 2) & (BTW_encoded.Year == 2016) & (BTW_encoded.meter == 0)]
building2 = building2.reset_index()
data2016 = building2[['Month', 'Day','QuarterDay','air_temperature', 'wind_speed', 'precip_depth_1_hr', 'cloud_coverage', 'meter_reading']]
building2 = BTW_test_encoded[(BTW_test_encoded.building_id == 2) & (BTW_test_encoded.Year == 2017) & (BTW_test_encoded.meter == 0)]
building2 = building2.reset_index()
data2017 = building2[['Month', 'Day','QuarterDay','air_temperature', 'wind_speed', 'precip_depth_1_hr', 'cloud_coverage', 'meter_reading']]
building2 = BTW_test_encoded[(BTW_test_encoded.building_id == 2) & (BTW_test_encoded.Year == 2018) & (BTW_test_encoded.meter == 0)]
building2 = building2.reset_index()
data2018 = building2[['Month', 'Day','QuarterDay','air_temperature', 'wind_speed', 'precip_depth_1_hr', 'cloud_coverage', 'meter_reading']]
```

In [208]:
```
std2018 = data2018["meter_reading"].std()
```

Further Data Understanding for Selected Building

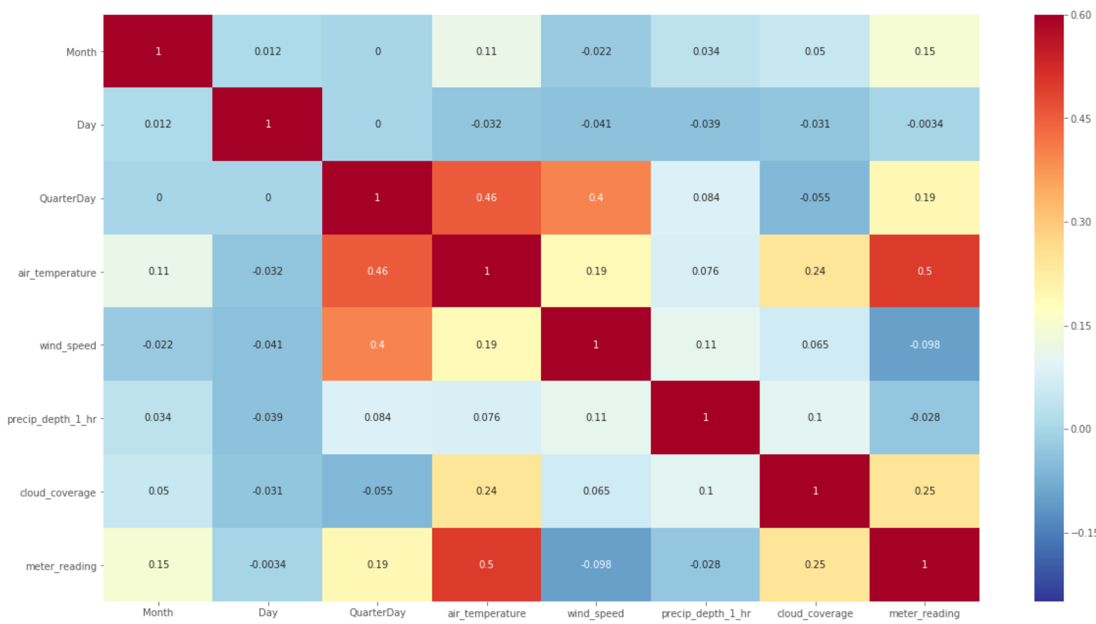
In [209]:
```
data2017.describe().astype(int)
```

Out[209]:

	Month	Day	QuarterDay	air_temperature	wind_speed	precip_depth_1_hr	cloud_c
count	1460	1460	1460	1460	1460	1460	
mean	6	15	2	22	3	1	
std	3	8	1	5	2	7	
min	1	1	1	3	0	-1	
25%	4	8	1	19	1	0	
50%	7	16	2	23	3	0	
75%	10	23	3	26	4	0	
max	12	31	4	34	20	150	

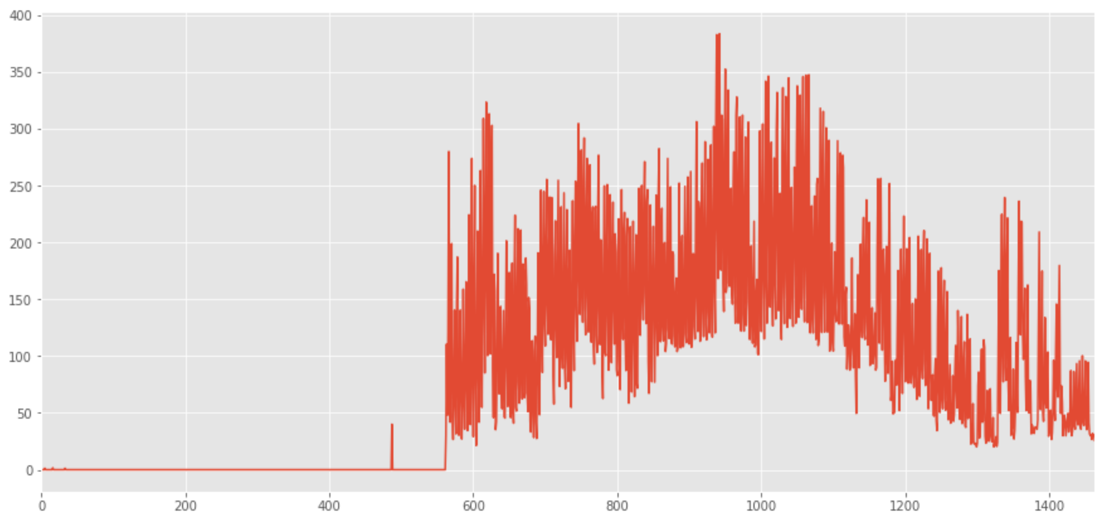
In [210]:
```
corrmat=data2017.corr()
plt.figure(figsize = (20,11))
sns.heatmap(corrmat,cmap=plt.cm.RdYlBu_r,vmin=-0.25,
            annot=True,vmax=0.6)
```

Out[210]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3ca9555ef0>



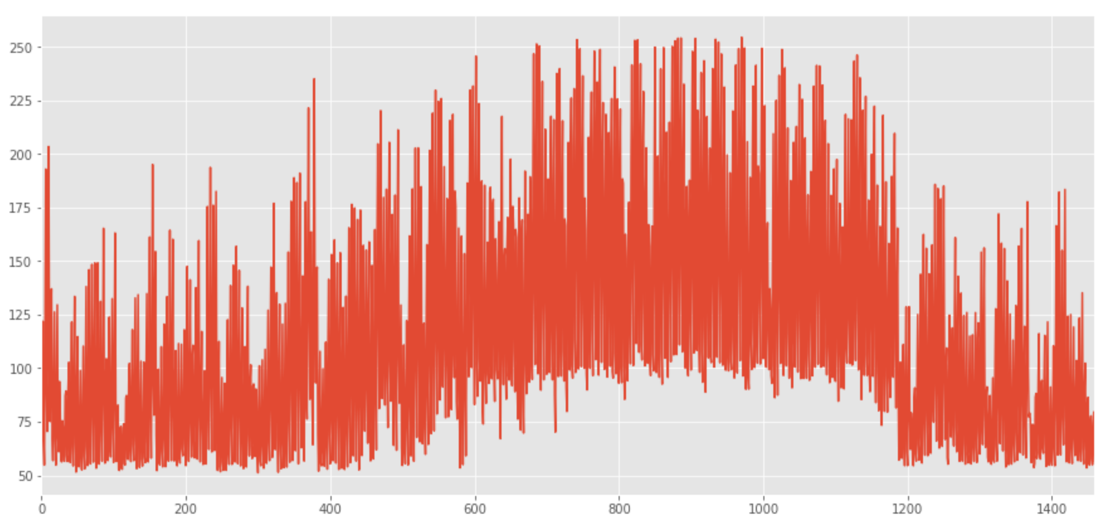
In [211]:
```
# plot data
fig, ax = plt.subplots(figsize=(15,7))
data2016['meter_reading'].plot()
```

Out[211]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3ce0a3a6d8>



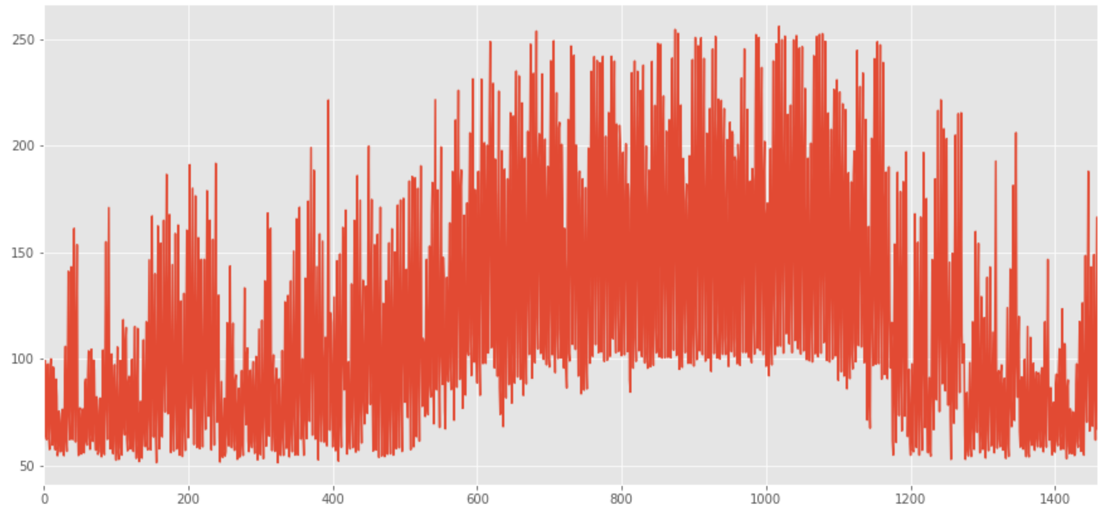
In [212]:
```
# plot data
fig, ax = plt.subplots(figsize=(15,7))
data2017['meter_reading'].plot()
```

Out[212]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3ca8f1ecc0>



In [213]:
```
# plot data
fig, ax = plt.subplots(figsize=(15,7))
data2018['meter_reading'].plot()
```

Out[213]: <matplotlib.axes._subplots.AxesSubplot at 0x7f34ac5a4128>



Convert to numpy and prepare for networks

In [214]:
```
np2016 = data2016.to_numpy()
np2017 = data2017.to_numpy()
np2018 = data2018.to_numpy()
np2018.shape
```

Out[214]: (1460, 8)

In [215]:
```
np2016 = normalize(np2016)
np2017 = normalize(np2017)
np2018 = normalize(np2018)
```

Now here is the data generator that we will use. It yields a tuple `(samples, targets)` where `samples` is one batch of input data and `targets` is the corresponding array of target temperatures. It takes the following arguments:

- `steps = 8` , i.e. our observations will be sampled at one data point per 48 hours.
- `delay = 56` , i.e. our targets will be one week in the future.
- `data` : The original array of floating point data.
- `lookback` : How many timesteps back should our input data go.
- `delay` : How many timesteps in the future should our target be.
- `min_index` and `max_index` : Indices in the `data` array that delimit which timesteps to draw from. This is useful for keeping a segment of the data for validation and another one for testing.
- `shuffle` : Whether to shuffle our samples or draw them in chronological order.
- `batch_size` : The number of samples per batch.
- `step` : The period, in timesteps, at which we sample data. We will set it 6 in order to draw one data point every hour.

In [216]:
```
#lookback = 168
lookback=336
step = 8
delay = 56
batch_size = 64
epochs = 40
steps_per_epoch=200

train_gen = generator(np2017,
                      lookback=lookback,
                      delay=delay,
                      min_index=0,
                      max_index=None,
                      shuffle=True,
                      step=step,
                      batch_size=batch_size)
val_gen = generator(np2018,
                    lookback=lookback,
                    delay=delay,
                    min_index=0,
                    max_index=None,
                    step=step,
                    batch_size=batch_size)
test_gen = generator(np2016,
                     lookback=lookback,
                     delay=delay,
                     min_index=0,
                     max_index=None,
                     step=step,
                     batch_size=batch_size)

# This is how many steps to draw from `val_gen`
# in order to see the whole validation set:
val_steps = (len(np2018) - lookback) // batch_size

# This is how many steps to draw from `test_gen`
# in order to see the whole test set:
test_steps = (len(np2016) - lookback) // batch_size
```

Non-machine learning baseline

In [217]:
```
np2018.shape
```

Out[217]:
```
(1460, 8)
```

In [218]:

```
def evaluate_naive_method():
    batch_maes = []
    for step in range(val_steps):
        samples, targets = next(val_gen)
        preds = samples[:, -1, 1]
        mae = np.mean(np.abs(preds - targets))
        batch_maes.append(mae)
    print("Mean Absolute Error Normalized: ", np.mean(batch_maes))
    return(np.mean(batch_maes))

mae = evaluate_naive_method()

print("Mean Absolute Error De-Normalized: ", std2018 * mae)
```

```
Mean Absolute Error Normalized:  1.732021998559289
Mean Absolute Error De-Normalized:  92.559603402836
```

A Basic dense layer

In [219]:

```
denseModel = Sequential()
denseModel.add(layers.Flatten(input_shape=(lookback // step, data20
17.shape[-1])))
denseModel.add(layers.Dense(32, activation='relu'))
denseModel.add(layers.Dense(1))
denseModel.compile(optimizer=RMSprop(), loss='mae')
denseModel.summary()
```

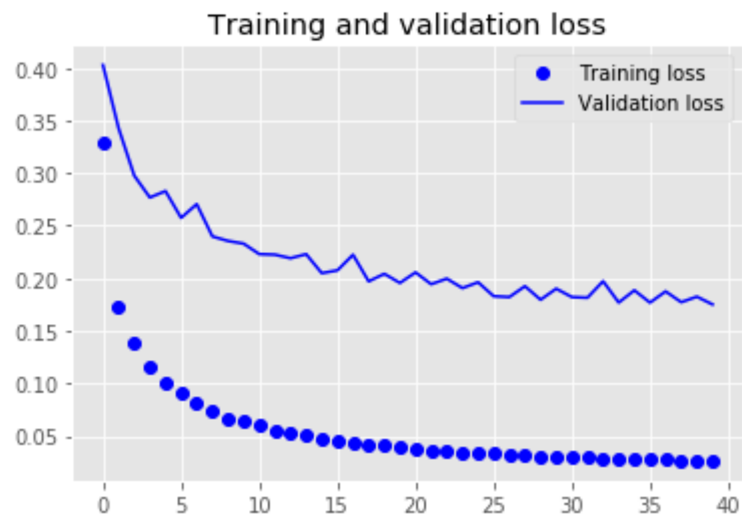
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_4 (Flatten)          (None, 336)               0
_________________________________________________________________
dense_15 (Dense)             (None, 32)                10784
_________________________________________________________________
dense_16 (Dense)             (None, 1)                 33
=================================================================
Total params: 10,817
Trainable params: 10,817
Non-trainable params: 0
_________________________________________________________________
```

In [220]:
```
## Compile and run the dense model
model_name = "DenseModel.h5"
checkpointer = ModelCheckpoint(filepath=model_name,
                               monitor = 'val_acc',
                               verbose=0,
                               save_best_only=True)

early_stopping = EarlyStopping(monitor='val_loss',
                          min_delta=0,
                          patience=5,
                          verbose=0, mode='auto')

denseHistory = denseModel.fit_generator(train_gen,
                              steps_per_epoch=steps_per_epoch,
                              epochs=epochs,
                              callbacks=[early_stopping],
                              validation_data=val_gen,
                              validation_steps=val_steps)
```

```
Epoch 1/40
200/200 [==============================] - 2s 12ms/step - loss: 
0.3283 - val_loss: 0.4027
Epoch 2/40
200/200 [==============================] - 1s 4ms/step - loss: 0.
1730 - val_loss: 0.3427
Epoch 3/40
200/200 [==============================] - 1s 4ms/step - loss: 0.
1391 - val_loss: 0.2976
Epoch 4/40
200/200 [==============================] - 1s 4ms/step - loss: 0.
1163 - val_loss: 0.2768
Epoch 5/40
200/200 [==============================] - 1s 4ms/step - loss: 0.
1008 - val_loss: 0.2829
Epoch 6/40
200/200 [==============================] - 1s 4ms/step - loss: 0.
0917 - val_loss: 0.2574
Epoch 7/40
200/200 [==============================] - 1s 4ms/step - loss: 0.
0813 - val_loss: 0.2704
Epoch 8/40
200/200 [==============================] - 1s 4ms/step - loss: 0.
0745 - val_loss: 0.2398
Epoch 9/40
200/200 [==============================] - 1s 4ms/step - loss: 0.
0661 - val_loss: 0.2355
Epoch 10/40
200/200 [==============================] - 1s 4ms/step - loss: 0.
0637 - val_loss: 0.2329
Epoch 11/40
200/200 [==============================] - 1s 4ms/step - loss: 0.
0601 - val_loss: 0.2229
Epoch 12/40
200/200 [==============================] - 1s 4ms/step - loss: 0.
0553 - val_loss: 0.2224
Epoch 13/40
200/200 [==============================] - 1s 4ms/step - loss: 0.
0524 - val_loss: 0.2190
Epoch 14/40
200/200 [==============================] - 1s 4ms/step - loss: 0.
0507 - val_loss: 0.2229
Epoch 15/40
200/200 [==============================] - 1s 4ms/step - loss: 0.
0471 - val_loss: 0.2048
Epoch 16/40
200/200 [==============================] - 1s 4ms/step - loss: 0.
0458 - val_loss: 0.2075
Epoch 17/40
200/200 [==============================] - 1s 4ms/step - loss: 0.
0429 - val_loss: 0.2223
Epoch 18/40
200/200 [==============================] - 1s 4ms/step - loss: 0.
0421 - val_loss: 0.1970
Epoch 19/40
```

In [221]:
```
plot_history(denseHistory)
```

```
<Figure size 1440x720 with 0 Axes>
```



In [222]:
```
print("Dense Mean Absolute Error Normal: ", min(denseHistory.histor
y['val_loss']))
print("Dense Mean Absolute Error De-Normalized: ", std2018 * min(de
nseHistory.history['val_loss']))
```

```
Dense Mean Absolute Error Normal:  0.17508860107730417
Dense Mean Absolute Error De-Normalized:  9.356770000353944
```

A first recurrent baseline

In [223]:
```
GRUBaselineModel = Sequential()
GRUBaselineModel.add(layers.GRU(32, input_shape=(None, data2017.sha
pe[-1])))
GRUBaselineModel.add(layers.Dense(1))
GRUBaselineModel.compile(optimizer=RMSprop(), loss='mae')
GRUBaselineModel.summary()
```

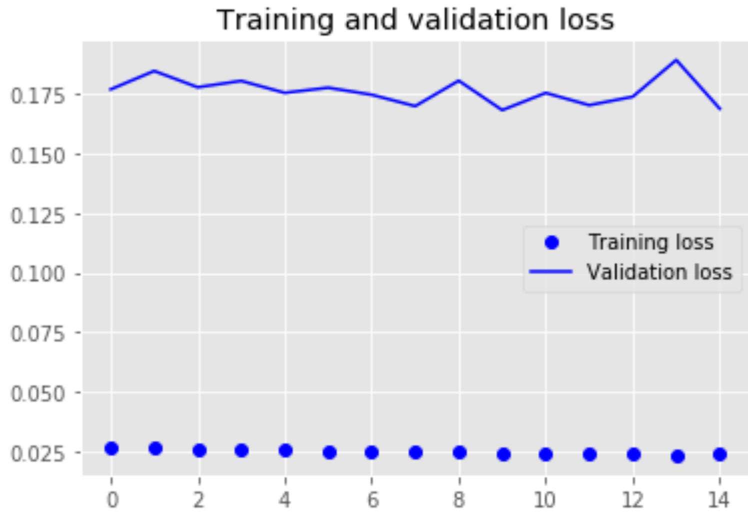
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
gru_13 (GRU)                 (None, 32)                3936
_________________________________________________________________
dense_17 (Dense)             (None, 1)                 33
=================================================================
Total params: 3,969
Trainable params: 3,969
Non-trainable params: 0
_________________________________________________________________
```

In [224]:
```
GRUBaselineHistory = denseModel.fit_generator(train_gen,
      steps_per_epoch=steps_per_epoch,
      epochs=epochs,
      callbacks=[early_stopping],
      validation_data=val_gen,
      validation_steps=val_steps)
```

```
Epoch 1/40
200/200 [==============================] - 2s 9ms/step - loss: 0.
0264 - val_loss: 0.1771
Epoch 2/40
200/200 [==============================] - 1s 4ms/step - loss: 0.
0263 - val_loss: 0.1848
Epoch 3/40
200/200 [==============================] - 1s 4ms/step - loss: 0.
0255 - val_loss: 0.1780
Epoch 4/40
200/200 [==============================] - 1s 4ms/step - loss: 0.
0257 - val_loss: 0.1807
Epoch 5/40
200/200 [==============================] - 1s 4ms/step - loss: 0.
0258 - val_loss: 0.1756
Epoch 6/40
200/200 [==============================] - 1s 4ms/step - loss: 0.
0252 - val_loss: 0.1778
Epoch 7/40
200/200 [==============================] - 1s 4ms/step - loss: 0.
0250 - val_loss: 0.1748
Epoch 8/40
200/200 [==============================] - 1s 4ms/step - loss: 0.
0248 - val_loss: 0.1700
Epoch 9/40
200/200 [==============================] - 1s 4ms/step - loss: 0.
0245 - val_loss: 0.1807
Epoch 10/40
200/200 [==============================] - 1s 4ms/step - loss: 0.
0242 - val_loss: 0.1684
Epoch 11/40
200/200 [==============================] - 1s 4ms/step - loss: 0.
0241 - val_loss: 0.1756
Epoch 12/40
200/200 [==============================] - 1s 4ms/step - loss: 0.
0239 - val_loss: 0.1704
Epoch 13/40
200/200 [==============================] - 1s 4ms/step - loss: 0.
0236 - val_loss: 0.1740
Epoch 14/40
200/200 [==============================] - 1s 4ms/step - loss: 0.
0232 - val_loss: 0.1894
Epoch 15/40
200/200 [==============================] - 1s 4ms/step - loss: 0.
0238 - val_loss: 0.1690
```

In [225]:
```
plot_history(GRUBaselineHistory)
```

```
<Figure size 1440x720 with 0 Axes>
```



In [226]:
```
print("GRU Baseline Mean Absolute Error Normal: ", min(GRUBaselineH
istory.history['val_loss']))
print("GRU Baseline Mean Absolute Error De-Normalized: ", std2018 *
min(GRUBaselineHistory.history['val_loss']))
```

```
GRU Baseline Mean Absolute Error Normal:  0.16836540313328013
GRU Baseline Mean Absolute Error De-Normalized:  8.99748095216907
2
```

Adding Recurrent Dropout

In [227]:
```
GRURecurrentModel = Sequential()
GRURecurrentModel.add(layers.GRU(32,
                      dropout=0.2,
                      recurrent_dropout=0.2,
                      input_shape=(None, data2017.shape[-1])))
GRURecurrentModel.add(layers.Dense(1))
GRURecurrentModel.compile(optimizer=RMSprop(), loss='mae')
GRURecurrentModel.summary()
```

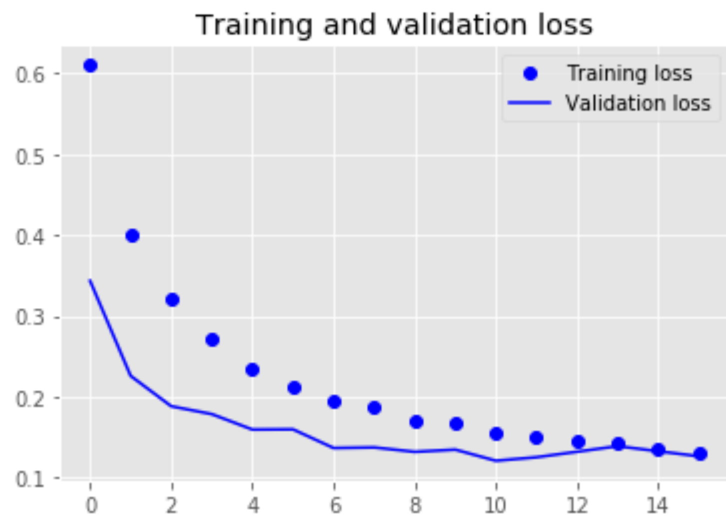
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
gru_14 (GRU)                 (None, 32)                3936
_________________________________________________________________
dense_18 (Dense)             (None, 1)                 33
=================================================================
Total params: 3,969
Trainable params: 3,969
Non-trainable params: 0
_________________________________________________________________
```

In [228]:
```
GRURecurrentHistory = GRURecurrentModel.fit_generator(train_gen,
      steps_per_epoch=steps_per_epoch,
      epochs=epochs,
      callbacks=[early_stopping],
      validation_data=val_gen,
      validation_steps=val_steps)
```

```
Epoch 1/40
200/200 [==============================] - 14s 69ms/step - loss:
0.6101 - val_loss: 0.3434
Epoch 2/40
200/200 [==============================] - 12s 58ms/step - loss:
0.4014 - val_loss: 0.2259
Epoch 3/40
200/200 [==============================] - 12s 58ms/step - loss:
0.3222 - val_loss: 0.1884
Epoch 4/40
200/200 [==============================] - 12s 60ms/step - loss:
0.2725 - val_loss: 0.1787
Epoch 5/40
200/200 [==============================] - 12s 58ms/step - loss:
0.2342 - val_loss: 0.1596
Epoch 6/40
200/200 [==============================] - 12s 58ms/step - loss:
0.2131 - val_loss: 0.1599
Epoch 7/40
200/200 [==============================] - 12s 58ms/step - loss:
0.1941 - val_loss: 0.1368
Epoch 8/40
200/200 [==============================] - 11s 57ms/step - loss:
0.1872 - val_loss: 0.1375
Epoch 9/40
200/200 [==============================] - 11s 57ms/step - loss:
0.1705 - val_loss: 0.1320
Epoch 10/40
200/200 [==============================] - 11s 57ms/step - loss:
0.1673 - val_loss: 0.1348
Epoch 11/40
200/200 [==============================] - 12s 58ms/step - loss:
0.1556 - val_loss: 0.1210
Epoch 12/40
200/200 [==============================] - 12s 58ms/step - loss:
0.1503 - val_loss: 0.1253
Epoch 13/40
200/200 [==============================] - 11s 56ms/step - loss:
0.1454 - val_loss: 0.1321
Epoch 14/40
200/200 [==============================] - 11s 57ms/step - loss:
0.1426 - val_loss: 0.1391
Epoch 15/40
200/200 [==============================] - 11s 57ms/step - loss:
0.1356 - val_loss: 0.1328
Epoch 16/40
200/200 [==============================] - 11s 57ms/step - loss:
0.1316 - val_loss: 0.1266
```

In [229]:
```
plot_history(GRURecurrentHistory)
```

<Figure size 1440x720 with 0 Axes>



In [230]:
```
print("GRU Recurrent Mean Absolute Error Normal: ", min(GRURecurrentHistory.history['val_loss']))
print("GRU Recurrent Mean Absolute Error De-Normalized: ", std2018 * min(GRURecurrentHistory.history['val_loss']))
```

```
GRU Recurrent Mean Absolute Error Normal:  0.12099657032419653
GRU Recurrent Mean Absolute Error De-Normalized:  6.466081014921715
```

Adding Layers with Dropout

In [231]:
```
## Adding Layers
GRUDeepModel = Sequential()
GRUDeepModel.add(layers.GRU(32,
                     dropout=0.1,
                     recurrent_dropout=0.5,
                     return_sequences=True,
                     input_shape=(None, data2017.shape[-1])))
GRUDeepModel.add(layers.GRU(64, activation='relu',
                     dropout=0.1,
                     recurrent_dropout=0.5,
                     return_sequences=True))
GRUDeepModel.add(layers.GRU(64, activation='relu',
                     dropout=0.1,
                     recurrent_dropout=0.5))
GRUDeepModel.add(layers.Dense(1))
GRUDeepModel.compile(optimizer=RMSprop(), loss='mae')
GRUDeepModel.summary()
```

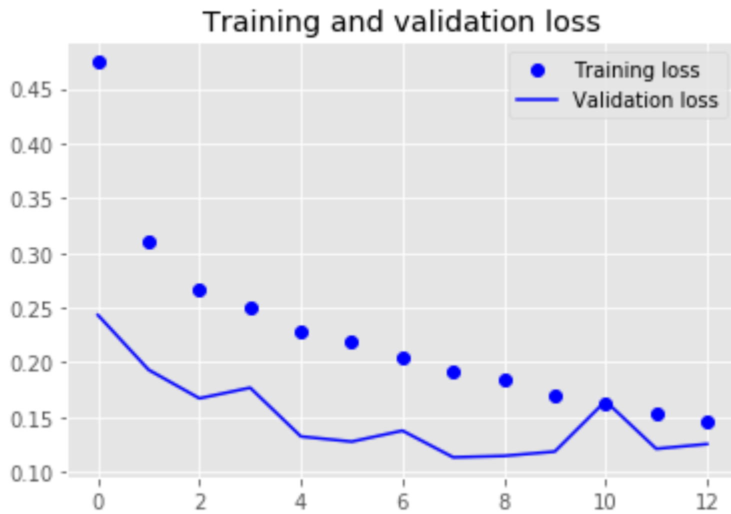
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
gru_15 (GRU)                 (None, None, 32)          3936
_________________________________________________________________
gru_16 (GRU)                 (None, None, 64)          18624
_________________________________________________________________
gru_17 (GRU)                 (None, 64)                24768
_________________________________________________________________
dense_19 (Dense)             (None, 1)                 65
=================================================================
Total params: 47,393
Trainable params: 47,393
Non-trainable params: 0
_________________________________________________________________
```

In [232]:
```
GRUDeepHistory = GRUDeepModel.fit_generator(train_gen,
      steps_per_epoch=steps_per_epoch,
      epochs=epochs,
      callbacks=[early_stopping],
      validation_data=val_gen,
      validation_steps=val_steps)
```

```
Epoch 1/40
200/200 [==============================] - 37s 187ms/step - loss:
0.4743 - val_loss: 0.2433
Epoch 2/40
200/200 [==============================] - 32s 161ms/step - loss:
0.3111 - val_loss: 0.1932
Epoch 3/40
200/200 [==============================] - 33s 165ms/step - loss:
0.2668 - val_loss: 0.1672
Epoch 4/40
200/200 [==============================] - 32s 162ms/step - loss:
0.2494 - val_loss: 0.1768
Epoch 5/40
200/200 [==============================] - 32s 160ms/step - loss:
0.2284 - val_loss: 0.1324
Epoch 6/40
200/200 [==============================] - 33s 164ms/step - loss:
0.2184 - val_loss: 0.1276
Epoch 7/40
200/200 [==============================] - 33s 165ms/step - loss:
0.2040 - val_loss: 0.1377
Epoch 8/40
200/200 [==============================] - 32s 160ms/step - loss:
0.1917 - val_loss: 0.1133
Epoch 9/40
200/200 [==============================] - 32s 162ms/step - loss:
0.1841 - val_loss: 0.1146
Epoch 10/40
200/200 [==============================] - 32s 161ms/step - loss:
0.1690 - val_loss: 0.1186
Epoch 11/40
200/200 [==============================] - 32s 161ms/step - loss:
0.1627 - val_loss: 0.1647
Epoch 12/40
200/200 [==============================] - 33s 167ms/step - loss:
0.1540 - val_loss: 0.1211
Epoch 13/40
200/200 [==============================] - 33s 166ms/step - loss:
0.1467 - val_loss: 0.1254
```

In [233]:
```
plot_history(GRUDeepHistory)
```

```
<Figure size 1440x720 with 0 Axes>
```



In [234]:
```
print("Baseline Mean Absolute Error Normal: ", mae)
print("Dense Mean Absolute Error Normal: ", min(denseHistory.histor
y['val_loss']))
print("GRU Baseline Mean Absolute Error Normal: ", min(GRUBaselineH
istory.history['val_loss']))
print("GRU Recurrent Mean Absolute Error Normal: ", min(GRURecurren
tHistory.history['val_loss']))
print("GRU Deep Mean Absolute Error Normal: ", min(GRUDeepHistory.h
istory['val_loss']))
print('')
print("Baseline Mean Absolute Error De-Normalized: ", std2018 * ma
e)
print("Dense Mean Absolute Error De-Normalized: ", std2018 * min(de
nseHistory.history['val_loss']))
print("GRU Baseline Mean Absolute Error De-Normalized: ", std2018 *
min(GRUBaselineHistory.history['val_loss']))
print("GRU Recurrent Mean Absolute Error De-Normalized: ", std2018
* min(GRURecurrentHistory.history['val_loss']))
print("GRU Deep Mean Absolute Error De-Normalized: ", std2018 * min
(GRUDeepHistory.history['val_loss']))
```

```
Baseline Mean Absolute Error Normal:  1.732021998559289
Dense Mean Absolute Error Normal:  0.17508860107730417
GRU Baseline Mean Absolute Error Normal:  0.16836540313328013
GRU Recurrent Mean Absolute Error Normal:  0.12099657032419653
GRU Deep Mean Absolute Error Normal:  0.11331309005618095

Baseline Mean Absolute Error De-Normalized:  92.559603402836
Dense Mean Absolute Error De-Normalized:  9.356770000353944
GRU Baseline Mean Absolute Error De-Normalized:  8.99748095216907
2
GRU Recurrent Mean Absolute Error De-Normalized:  6.4660810149217
15
GRU Deep Mean Absolute Error De-Normalized:  6.055474286512606
```

In [235]:
```
# reshape input to be [samples, time steps, features]
#np2016reshape = np.reshape(np2016, (np2016.shape[0], 1, np2016.sha
pe[1]))
```

In [236]:
```
#prediction = GRUDeepModel.predict(np2016reshape)
```

In [237]:
```
#y = prediction[0:]
#x = np.arange(0,len(prediction))
```

In [238]:
```
#plt.figure(figsize=(20, 10))
#plt.style.use('ggplot') # Grammar of Graphics plots
#plt.figure()
#plt.plot(x, y, label='prediction')
#plt.title('Prediction')
#plt.legend()
#plt.show()
#plt.close()
```

In [ ]: