In [1]:
```
import datetime
import pandas as pd
import datetime as dt
from pandas_datareader import data
import matplotlib.pyplot as plt
import os
import numpy as np
import tensorflow as tf # This code has been tested with TensorFlow 1.6
from sklearn.preprocessing import MinMaxScaler
```

```
C:\Users\vijayalakshmi\Anaconda3\lib\site-packages\h5py\__init__.py:36: FutureWa
rning: Conversion of the second argument of issubdtype from `float` to `np.float
ing` is deprecated. In future, it will be treated as `np.float64 == np.dtype(flo
at).type`.
  from ._conv import register_converters as _register_converters
```

In [2]:
```
df = pd.read_csv("C:\\Users\\vijayalakshmi\\Documents\\STOCK PRICE DATA.csv")
```

In [4]:
```
print(df.head())
df.describe()
```

```
           isin  insertion_datetime         datetime1   Openn     high  \
0  INE545A01016                 NaN  01-01-2018 09:15  2419.0  2436.00
1  INE545A01016                 NaN           16:00.0  2435.0  2440.00
2  INE545A01016                 NaN           17:00.0  2439.7  2439.70
3  INE545A01016                 NaN           18:00.0  2420.0  2424.50
4  INE545A01016                 NaN           19:00.0  2395.0  2413.95

       low   closse  volume  open_interes
0  2408.90  2435.25   10281           NaN
1  2426.45  2439.00   13315           NaN
2  2422.10  2424.95    3760           NaN
3  2386.50  2395.00   11653           NaN
4  2394.70  2409.85    7744           NaN
```

Out[4]:

	insertion_datetime	Openn	high	low	closse	volume	open_
count	0.0	6176.000000	6176.000000	6176.000000	6176.000000	6176.000000	0.0
mean	NaN	2671.088212	2674.293321	2668.373551	2671.391491	1460.788698	NaN
std	NaN	96.031803	96.488606	95.755028	96.081734	2348.309972	NaN
min	NaN	2395.000000	2413.950000	2386.500000	2395.000000	1.000000	NaN
25%	NaN	2594.237500	2596.900000	2592.000000	2594.487500	315.750000	NaN
50%	NaN	2658.000000	2660.675000	2655.100000	2658.525000	732.000000	NaN
75%	NaN	2763.412500	2767.700000	2760.212500	2763.950000	1599.500000	NaN
max	NaN	2890.650000	2892.000000	2883.300000	2890.650000	39241.000000	NaN

In [5]:
```
df1=pd.DataFrame(df)
data=df1.drop(['insertion_datetime','open_interes','volume','isin'],axis=1)
data.head()
```

Out[5]:

	datetime1	Openn	high	low	closse
0	01-01-2018 09:15	2419.0	2436.00	2408.90	2435.25
1	16:00.0	2435.0	2440.00	2426.45	2439.00
2	17:00.0	2439.7	2439.70	2422.10	2424.95
3	18:00.0	2420.0	2424.50	2386.50	2395.00
4	19:00.0	2395.0	2413.95	2394.70	2409.85

In [8]:
```
df1=data.rename(columns={'datetime1':'Date','Openn':'open','closse':'close'})
df1.head()
df2=df1.drop_duplicates(keep=False,inplace=False)
df2.shape
```

Out[8]: (5160, 5)

In [9]:
```
high_prices = df2.loc[:,'high'].values
low_prices = df2.loc[:,'low'].values
mid_prices = (high_prices+low_prices)/2.0
print(mid_prices)
```

```
[2422.45  2433.225 2430.9   ... 2605.    2607.425 2606.05 ]
```

In [10]:
```
train_data = mid_prices[:4000]
test_data = mid_prices[4000:]
```

In [11]:
```
scaler = MinMaxScaler()
train_data = train_data.reshape(-1,1)
test_data = test_data.reshape(-1,1)
```

In [12]:
```
smoothing_window_size = 2500
for di in range(0,2000,smoothing_window_size):
    scaler.fit(train_data[di:di+smoothing_window_size,:])
    train_data[di:di+smoothing_window_size,:] = scaler.transform(train_data[di:di+s
moothing_window_size,:])
```

In [13]:
```
scaler.fit(train_data[di+smoothing_window_size:,:])
train_data[di+smoothing_window_size:,:] = scaler.transform(train_data[di+smoothing_
window_size:,:])
```

In [14]:
```
train_data = train_data.reshape(-1)
test_data = scaler.transform(test_data).reshape(-1)
```

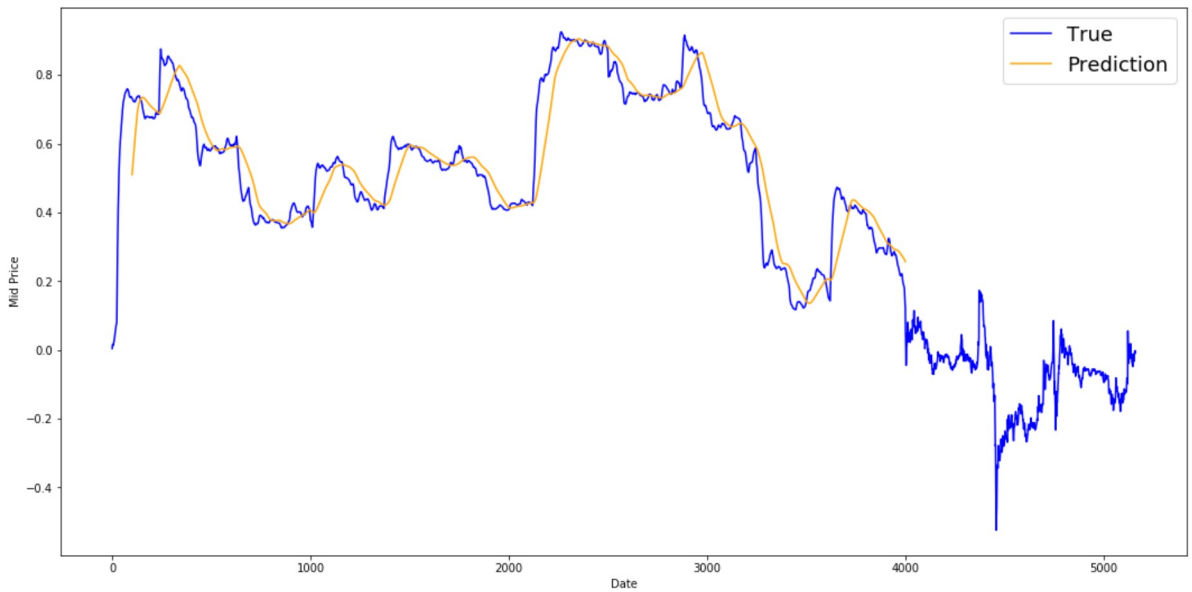
In [15]:
```
EMA = 0.0
gamma = 0.1
for ti in range(4000):
  EMA = gamma*train_data[ti] + (1-gamma)*EMA
  train_data[ti] = EMA
all_mid_data = np.concatenate([train_data,test_data],axis=0)
```

In [24]:
```
window_size = 100
N = train_data.size
std_avg_predictions = []
std_avg_x = []
mse_errors = []

for pred_idx in range(window_size,N):

    if pred_idx >= N:
        date = df1.datetime.strptime(k, '%y-%m-%d').date() + dt.timedelta(days=1)
    else:
        date = df1.loc[pred_idx,'Date']

    std_avg_predictions.append(np.mean(train_data[pred_idx-window_size:pred_idx]))
    mse_errors.append((std_avg_predictions[-1]-train_data[pred_idx])**2)
    std_avg_x.append(date)
```

In [17]:
```
print('MSE error for standard averaging: %.5f'%(0.5*np.mean(mse_errors)))
```

```
MSE error for standard averaging: 0.00238
```

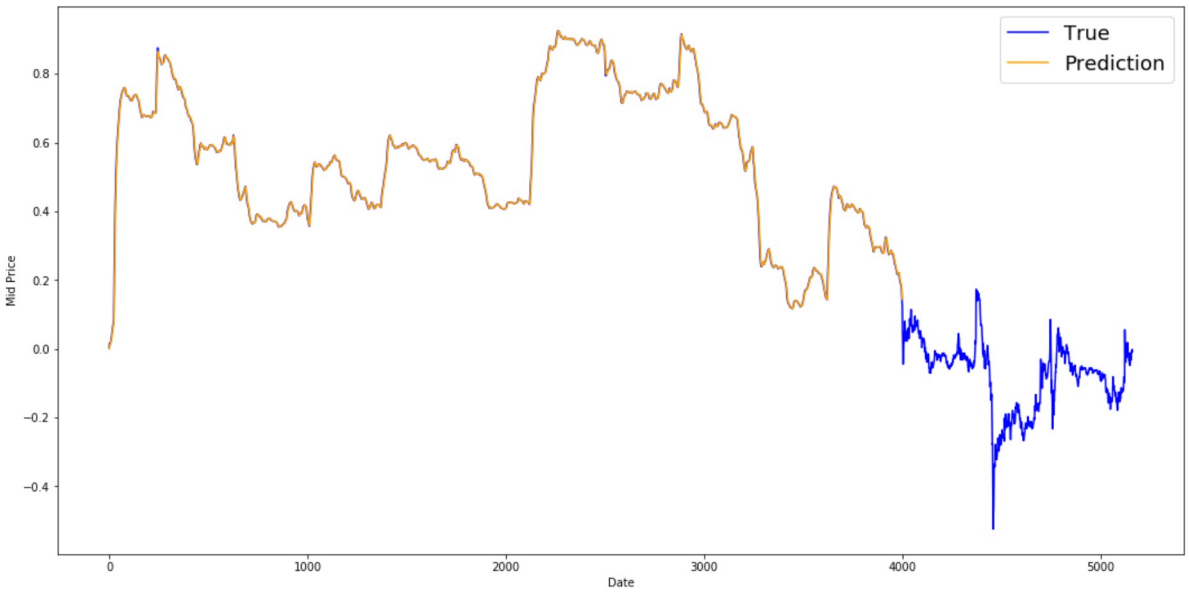
In [18]:
```
plt.figure(figsize = (18,9))
plt.plot(range(df2.shape[0]),all_mid_data,color='b',label='True')
plt.plot(range(window_size,N),std_avg_predictions,color='orange',label='Prediction'
)

plt.xlabel('Date')
plt.ylabel('Mid Price')
plt.legend(fontsize=18)
plt.show()
```



In [25]:
```
window_size = 100
N = train_data.size

run_avg_predictions = []
run_avg_x = []

mse_errors = []

running_mean = 0.0
run_avg_predictions.append(running_mean)

decay = 0.5
for pred_idx in range(1,N):

    running_mean = running_mean*decay + (1.0-decay)*train_data[pred_idx-1]
    run_avg_predictions.append(running_mean)
    mse_errors.append((run_avg_predictions[-1]-train_data[pred_idx])**2)
    run_avg_x.append(date)

print('MSE error for EMA averaging: %.5f'%(0.5*np.mean(mse_errors)))
```

```
MSE error for EMA averaging: 0.00003
```

In [27]:
```
plt.figure(figsize = (18,9))
plt.plot(range(df2.shape[0]),all_mid_data,color='b',label='True')
plt.plot(range(0,N),run_avg_predictions,color='orange', label='Prediction')
#plt.xticks(range(0,df.shape[0],50),df['Date'].loc[::50],rotation=45)
plt.xlabel('Date')
plt.ylabel('Mid Price')
plt.legend(fontsize=18)
plt.show()
```



In [16]:
```
class DataGeneratorSeq(object):

    def __init__(self,prices,batch_size,num_unroll):
        self._prices = prices
        self._prices_length = len(self._prices) - num_unroll
        self._batch_size = batch_size
        self._num_unroll = num_unroll
        self._segments = self._prices_length //self._batch_size
        self._cursor = [offset * self._segments for offset in range(self._batch_siz
e)]

    def next_batch(self):

        batch_data = np.zeros((self._batch_size),dtype=np.float32)
        batch_labels = np.zeros((self._batch_size),dtype=np.float32)

        for b in range(self._batch_size):
            if self._cursor[b]+1>=self._prices_length:
                #self._cursor[b] = b * self._segments
                self._cursor[b] = np.random.randint(0,(b+1)*self._segments)
                batch_data[b] = self._prices[self._cursor[b]]
            batch_labels[b]= self._prices[self._cursor[b]+np.random.randint(0,5)]

            self._cursor[b] = (self._cursor[b]+1)%self._prices_length

        return batch_data,batch_labels

    def unroll_batches(self):
         unroll_data,unroll_labels = [],[]
         init_data, init_label = None,None
         for ui in range(self._num_unroll):
            data, labels = self.next_batch()

            unroll_data.append(data)
            unroll_labels.append(labels)

         return unroll_data, unroll_labels

    def reset_indices(self):
        for b in range(self._batch_size):
            self._cursor[b] = np.random.randint(0,min((b+1)*self._segments,self._pr
ices_length-1))



dg = DataGeneratorSeq(train_data,5,5)
u_data, u_labels = dg.unroll_batches()
for ui,(dat,lbl) in enumerate(zip(u_data,u_labels)):
    print('\n\nUnrolled index %d'%ui)
    dat_ind = dat
    lbl_ind = lbl
    print('\tInputs: ',dat )
    print('\n\tOutput:',lbl)
```

```
Unrolled index 0
        Inputs:  [0. 0. 0. 0. 0.]

        Output: [0.01053966 0.3778733  0.5515337  0.88937473 0.5622578 ]


Unrolled index 1
        Inputs:  [0. 0. 0. 0. 0.]

        Output: [0.01053966 0.3781185  0.5533732  0.8939682  0.55666816]


Unrolled index 2
        Inputs:  [0. 0. 0. 0. 0.]

        Output: [0.01294788 0.3784732  0.5533732  0.88937473 0.5401974 ]


Unrolled index 3
        Inputs:  [0. 0. 0. 0. 0.]

        Output: [0.01368118 0.37886414 0.5544662  0.88937473 0.52732414]


Unrolled index 4
        Inputs:  [0. 0. 0. 0. 0.]

        Output: [0.01703945 0.3790584  0.5533732  0.88537586 0.52336437]
```

In [17]:
```
tf.reset_default_graph()

D = 1 # Dimensionality of the data. Since your data is 1-D this would be 1
num_unrollings = 50 # Number of time steps you look into the future.
batch_size = 500 # Number of samples in a batch
num_nodes = [200,200,150] # Number of hidden nodes in each layer of the deep LSTM s
tack we're using
n_layers = len(num_nodes) # number of layers
dropout = 0.2 # dropout amount


w = tf.get_variable('w',shape=[num_nodes[-1],1], initializer=tf.contrib.layers.xavi
er_initializer())
b = tf.get_variable('b',initializer=tf.random_uniform([1],-0.1,0.1))

train_inputs, train_outputs = [],[]

# You unroll the input over time defining placeholders for each time step
for ui in range(num_unrollings):
    train_inputs.append(tf.placeholder(tf.float32, shape=[batch_size,D],name='train
_inputs_%d'%ui))
    train_outputs.append(tf.placeholder(tf.float32, shape=[batch_size,1], name = 't
rain_outputs_%d'%ui))
```

```
WARNING: The TensorFlow contrib module will not be included in TensorFlow 2.0.
For more information, please see:
  * https://github.com/tensorflow/community/blob/master/rfcs/20180907-contrib-su
nset.md
  * https://github.com/tensorflow/addons
If you depend on functionality not listed there, please file an issue.

WARNING:tensorflow:From C:\Users\vijayalakshmi\Anaconda3\lib\site-packages\tenso
rflow\python\framework\op_def_library.py:263: colocate_with (from tensorflow.pyt
hon.framework.ops) is deprecated and will be removed in a future version.
Instructions for updating:
Colocations handled automatically by placer.
```

In [18]:
```
tf.reset_default_graph()


train_inputs, train_outputs = [],[]
for ui in range(num_unrollings):
    train_inputs.append(tf.placeholder(tf.float32, shape=[batch_size,D],name='train
_inputs_%d'%ui))
    train_outputs.append(tf.placeholder(tf.float32, shape=[batch_size,1], name = 't
rain_outputs_%d'%ui))

lstm_cells = [
    tf.contrib.rnn.LSTMCell(num_units=num_nodes[li],
                            state_is_tuple=True,
                            initializer= tf.contrib.layers.xavier_initializer()
                           )
 for li in range(n_layers)]

drop_lstm_cells = [tf.contrib.rnn.DropoutWrapper(
    lstm, input_keep_prob=1.0,output_keep_prob=1.0-dropout, state_keep_prob=1.0-dro
pout
) for lstm in lstm_cells]
drop_multi_cell = tf.contrib.rnn.MultiRNNCell(drop_lstm_cells)
multi_cell = tf.contrib.rnn.MultiRNNCell(lstm_cells)

w = tf.get_variable('w',shape=[num_nodes[-1], 1], initializer=tf.contrib.layers.xav
ier_initializer())
b = tf.get_variable('b',initializer=tf.random_uniform([1],-0.1,0.1))
c, h = [],[]
initial_state = []
for li in range(n_layers):
  c.append(tf.Variable(tf.zeros([batch_size, num_nodes[li]]), trainable=False))
  h.append(tf.Variable(tf.zeros([batch_size, num_nodes[li]]), trainable=False))
  initial_state.append(tf.contrib.rnn.LSTMStateTuple(c[li], h[li]))

# Do several tensor transofmations, because the function dynamic_rnn requires the o
utput to be of
# a specific format. Read more at: https://www.tensorflow.org/api_docs/python/tf/nn
/dynamic_rnn
all_inputs = tf.concat([tf.expand_dims(t,0) for t in train_inputs],axis=0)

# all_outputs is [seq_length, batch_size, num_nodes]
all_lstm_outputs, state = tf.nn.dynamic_rnn(
    drop_multi_cell, all_inputs, initial_state=tuple(initial_state),time_major = Tr
ue, dtype=tf.float32)

all_lstm_outputs = tf.reshape(all_lstm_outputs, [batch_size*num_unrollings,num_node
s[-1]])

all_outputs = tf.nn.xw_plus_b(all_lstm_outputs,w,b)

split_outputs = tf.split(all_outputs,num_unrollings,axis=0)
```

```
WARNING:tensorflow:From <ipython-input-18-5f9d86ee06f8>:14: LSTMCell.__init__ (f
rom tensorflow.python.ops.rnn_cell_impl) is deprecated and will be removed in a
future version.
Instructions for updating:
This class is equivalent as tf.keras.layers.LSTMCell, and will be replaced by th
at in Tensorflow 2.0.
WARNING:tensorflow:From <ipython-input-18-5f9d86ee06f8>:19: MultiRNNCell.__init_
_ (from tensorflow.python.ops.rnn_cell_impl) is deprecated and will be removed i
n a future version.
Instructions for updating:
This class is equivalent as tf.keras.layers.StackedRNNCells, and will be replace
d by that in Tensorflow 2.0.
WARNING:tensorflow:From <ipython-input-18-5f9d86ee06f8>:37: dynamic_rnn (from te
nsorflow.python.ops.rnn) is deprecated and will be removed in a future version.
Instructions for updating:
Please use `keras.layers.RNN(cell)`, which is equivalent to this API
WARNING:tensorflow:From C:\Users\vijayalakshmi\Anaconda3\lib\site-packages\tenso
rflow\python\ops\rnn_cell_impl.py:1259: calling dropout (from tensorflow.python.
ops.nn_ops) with keep_prob is deprecated and will be removed in a future version
.
Instructions for updating:
Please use `rate` instead of `keep_prob`. Rate should be set to `rate = 1 - keep
_prob`.
```

In [19]:

```
print('Defining training Loss')
loss = 0.0

with tf.control_dependencies([tf.assign(c[li], state[li][0]) for li in range(n_laye
rs)]+
                             [tf.assign(h[li], state[li][1]) for li in range(n_laye
rs)]):
  for ui in range(num_unrollings):
    loss += tf.reduce_mean(0.5*(split_outputs[ui]-train_outputs[ui])**2)

print('Learning rate decay operations')
global_step = tf.Variable(0, trainable=False)
inc_gstep = tf.assign(global_step,global_step + 1)
tf_learning_rate = tf.placeholder(shape=None,dtype=tf.float32)
tf_min_learning_rate = tf.placeholder(shape=None,dtype=tf.float32)
learning_rate = tf.maximum(
    tf.train.exponential_decay(tf_learning_rate, global_step, decay_steps=1, decay_
rate=0.5, staircase=True),
    tf_min_learning_rate)
print('TF Optimization operations')
optimizer = tf.train.AdamOptimizer(learning_rate)
gradients, v = zip(*optimizer.compute_gradients(loss))
gradients, _ = tf.clip_by_global_norm(gradients, 5.0)
optimizer = optimizer.apply_gradients(
    zip(gradients, v))

print('\tAll done')
```

```
Defining training Loss
Learning rate decay operations
TF Optimization operations
        All done
```

In [20]:
```
sample_inputs = tf.placeholder(tf.float32, shape=[1,D])

# Maintaining LSTM state for prediction stage
sample_c, sample_h, initial_sample_state = [],[],[]
for li in range(n_layers):
  sample_c.append(tf.Variable(tf.zeros([1, num_nodes[li]]), trainable=False))
  sample_h.append(tf.Variable(tf.zeros([1, num_nodes[li]]), trainable=False))
  initial_sample_state.append(tf.contrib.rnn.LSTMStateTuple(sample_c[li],sample_h[li]))

reset_sample_states = tf.group(*[tf.assign(sample_c[li],tf.zeros([1, num_nodes[li]])) for li in range(n_layers)],
                               *[tf.assign(sample_h[li],tf.zeros([1, num_nodes[li]])) for li in range(n_layers)])

sample_outputs, sample_state = tf.nn.dynamic_rnn(multi_cell, tf.expand_dims(sample_inputs,0),
                                 initial_state=tuple(initial_sample_state),
                                 time_major = True,
                                 dtype=tf.float32)
with tf.control_dependencies([tf.assign(sample_c[li],sample_state[li][0]) for li in range(n_layers)]+
                             [tf.assign(sample_h[li],sample_state[li][1]) for li in range(n_layers)]):
  sample_prediction = tf.nn.xw_plus_b(tf.reshape(sample_outputs,[1,-1]), w, b)

print('\tAll done')
```

```
	All done
```

In [21]:
```
epochs = 30
valid_summary = 1 # Interval you make test predictions

n_predict_once = 50 # Number of steps you continously predict for

train_seq_length = train_data.size # Full length of the training data

train_mse_ot = [] # Accumulate Train losses
test_mse_ot = [] # Accumulate Test loss
predictions_over_time = [] # Accumulate predictions

session = tf.InteractiveSession()

tf.global_variables_initializer().run()

# Used for decaying learning rate
```

In [26]:
```
loss_nondecrease_count = 0
loss_nondecrease_threshold = 2 # If the test error hasn't increased in this many st
eps, decrease learning rate

print('Initialized')
average_loss = 0

# Define data generator
data_gen = DataGeneratorSeq(train_data,batch_size,num_unrollings)

x_axis_seq = []

# Points you start your test predictions from
test_points_seq = np.arange(4000,5000,50).tolist()
```

```
Initialized
```

In [25]:
```
test_points_seq = np.arange(4000,5000,50).tolist()
print(test_points_seq )
```

```
[4000, 4050, 4100, 4150, 4200, 4250, 4300, 4350, 4400, 4450, 4500, 4550, 4600, 4
650, 4700, 4750, 4800, 4850, 4900, 4950]
```

In [27]:
```
for ep in range(epochs):

    # ======================== Training ====================================
    for step in range(train_seq_length//batch_size):

        u_data, u_labels = data_gen.unroll_batches()

        feed_dict = {}
        for ui,(dat,lbl) in enumerate(zip(u_data,u_labels)):
            feed_dict[train_inputs[ui]] = dat.reshape(-1,1)
            feed_dict[train_outputs[ui]] = lbl.reshape(-1,1)

        feed_dict.update({tf_learning_rate: 0.0001, tf_min_learning_rate:0.000001})
        _, l = session.run([optimizer, loss], feed_dict=feed_dict)

        average_loss += l
    if (ep+1) % valid_summary == 0:

      average_loss = average_loss/(valid_summary*(train_seq_length//batch_size))

      # The average loss
      if (ep+1)%valid_summary==0:
        print('Average loss at step %d: %f' % (ep+1, average_loss))

      train_mse_ot.append(average_loss)

      average_loss = 0 # reset loss

      predictions_seq = []

      mse_test_loss_seq = []
    for w_i in test_points_seq:
        mse_test_loss = 0.0
        our_predictions = []

        if (ep+1)-valid_summary==0:
          # Only calculate x_axis values in the first validation epoch
          x_axis=[]

        # Feed in the recent past behavior of stock prices
        # to make predictions from that point onwards
        for tr_i in range(w_i-num_unrollings+1,w_i-1):
          current_price = all_mid_data[tr_i]

          feed_dict[sample_inputs] = np.array(current_price).reshape(1,1)
          _ = session.run(sample_prediction,feed_dict=feed_dict)
          feed_dict = {}

        current_price = all_mid_data[w_i-1]

        feed_dict[sample_inputs] = np.array(current_price).reshape(1,1)
        for pred_i in range(n_predict_once):

          pred = session.run(sample_prediction,feed_dict=feed_dict)

          our_predictions.append(np.asscalar(pred))

          feed_dict[sample_inputs] = np.asarray(pred).reshape(-1,1)

          if (ep+1)-valid_summary==0:
            # Only calculate x_axis values in the first validation epoch
            x_axis.append(w_i+pred_i)

        mse_test_loss += 0.5*(pred-all_mid_data[w_i+pred_i])**2
```

```
Average loss at step 1: 6.731521
Average loss at step 2: 2.566938
Average loss at step 3: 2.715710
Average loss at step 4: 1.403398
Average loss at step 5: 1.385928
Average loss at step 6: 1.325942
Average loss at step 7: 1.179651
Average loss at step 8: 1.154204
Average loss at step 9: 1.178433
Average loss at step 10: 1.014098
Average loss at step 11: 1.051108
Average loss at step 12: 1.088499
Average loss at step 13: 1.094963
Average loss at step 14: 1.078799
Average loss at step 15: 1.267820
Average loss at step 16: 1.325193
Average loss at step 17: 1.183821
Average loss at step 18: 1.143118
Average loss at step 19: 1.111487
Average loss at step 20: 1.015280
Average loss at step 21: 1.022394
Average loss at step 22: 1.106518
Average loss at step 23: 1.062111
Average loss at step 24: 1.203770
Average loss at step 25: 1.307144
Average loss at step 26: 1.174281
Average loss at step 27: 1.130367
Average loss at step 28: 1.120766
Average loss at step 29: 1.088690
Average loss at step 30: 1.031407
        Test MSE: 0.07764
        Finished Predictions
```

In [42]:
```
best_prediction_epoch = 25 # replace this with the epoch that you got the best resu
lts when running the plotting code

plt.figure(figsize = (18,18))
plt.subplot(2,1,1)
plt.plot(range(df2.shape[0]),all_mid_data,color='b')
start_alpha = 0.15
alpha  = np.arange(start_alpha,1.1,(1.0-start_alpha)/len(predictions_over_time[::2]
))
for p_i,p in enumerate(predictions_over_time[::2]):
    for xval,yval in zip(x_axis_seq,p):
        plt.plot(xval,yval,color='r',alpha=alpha[p_i])
plt.title('Evolution of Test Predictions Over Time',fontsize=18)
plt.xlabel('Date',fontsize=18)
plt.ylabel('Mid Price',fontsize=18)
plt.xlim(4000,5100)
```

Out[42]: (4000, 5100)