Traditional neural networks suffer from the vanishing gradient problem; that is, sensitivity decays exponentially over time. This is bad news to gather significance from sequential, time dependant data. To fix the problem, one can use LTSM, or GRU, or such neural network derivitives employing long term dependencies to keep the big picture in mind.

In [6]:

```
import pandas as pd
import numpy as np
from keras.layers.core import Dense, Activation, Dropout,ActivityRe
gularization
from keras.layers import Dense, D
ropout, Embedding, LSTM, Bidirectional
from keras import regularizers
from keras.models import Sequential
import time
from sklearn.metrics import mean_squared_error
from matplotlib import pyplot
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
from math import sqrt
from pandas import DataFrame
from pandas import concat
from pandas import read_csv
from sklearn.preprocessing import LabelEncoder
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Conv1D, MaxPooling1D, LeakyReLU, PReLU
from keras.layers import GRU,CuDNNGRU
import matplotlib.pyplot as plt
```

```
/opt/anaconda3/lib/python3.6/site-packages/h5py/__in
it__.py:36: FutureWarning: Conversion of the second
argument of issubdtype from `float` to `np.floating`
is deprecated. In future, it will be treated as `np.
float64 == np.dtype(float).type`.
  from ._conv import register_converters as _registe
r_converters
Using TensorFlow backend.
```

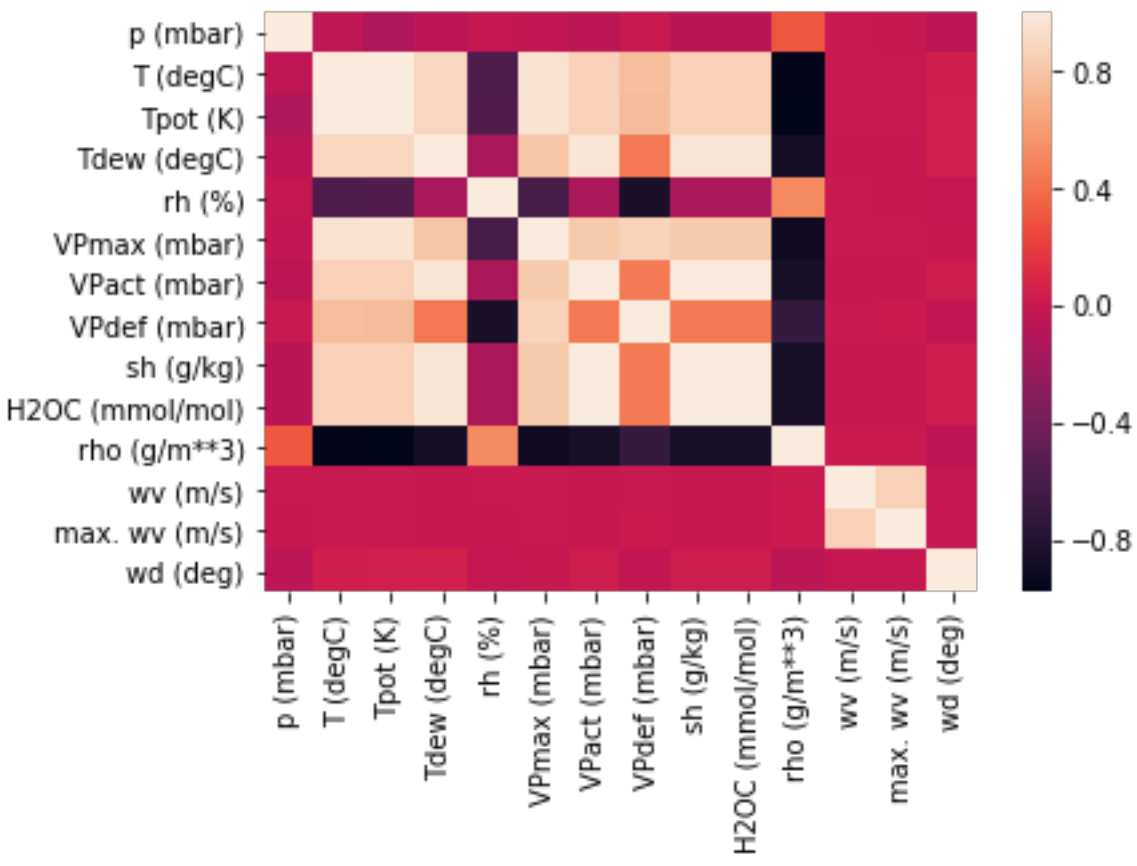
This is essentially a multivariate regression analysis, so correlations between the variables are important, for which I use Seaborn to produce one heatmap for simple visuallization. It seems that most are eighter highly negatively or positively correlated, making the predictive modelling valid.

In [7]:

```
import seaborn as sns
path='climate_hour.csv'
dataset = read_csv(path,index_col=0)
corr = dataset.corr()
sns.heatmap(corr, xticklabels=corr.columns.values,yticklabels=corr.
columns.values)
```

Out[7]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x3fff0f8f
8e10>
```



Every 24 hr is considered a composite unit of variable that predicts the outcome of the temperature at the 25th hour. here we create a window size of 24 then slide it over the data to create such 24 hr units.

window_size = 24 series_s = dataset.copy() for i in range(window_size-1): dataset = pd.concat([dataset, series_s.shift(-(i+1))], axis = 1) dataset.dropna(axis=0, inplace=True)

The beginning and end times and dates were specified by the project instructions. Here each dataset is parsed appropriately accordingly.

In [10]:

```
#parse the datasets according to dates
df = pd.DataFrame(dataset)
index=pd.to_datetime(df.index)
mask = (index <='31.12.2014 22:00:00')
x_train=df.loc[mask]
mask2=(index>='31.12.2014 00:00:00')&(index<='31.12.2016 23:00:00')
x_test=df.loc[mask2]
df2=DataFrame(dataset)
index2=pd.to_datetime(df2.index)
mask3=(index2>='02.01.2009 01:00:00')&(index2<='31.12.2014 23:00:00
')
thing1=df2.loc[mask3]
y_train=thing1.iloc[:,1]
mask4=(index2>='01.01.2015 00:00:00')&(index2<='01.01.2017 00:00:00
')
thing2=df2.loc[mask4]
y_test=thing2.iloc[:,1]
```

After splicing the data in such manner, y_train and y_test came up short, so I concatenated the beginning of each set to its end to correspond natural seasonal cycles.

In [16]:

```
#Concatenate the missing values end the tail end with the beginnin
g of the set to match the x_train and x_test
first=y_test.values[:24]
y_test=y_test.values
y_test=np.concatenate((y_test,first),axis=0)
#then resize in
to verticle vectors
y_test=y_test.reshape(y_test.shape[0],1)
copy=y_test
two=y_train.values[:743]
y_train=y_train.values
y_train=np.concatenate((y_train,two),axis=0) #then resize into vert
icle vectors
y_train=y_train.reshape(y_train.shape[0],1)
```

Since the dataset has such wide ranging values, normalization between zero and one is required to make predictive modelling possible.

In [17]:

```
#normalize y_train and y_test
scaler = MinMaxScaler(feature_range=(0, 1))
y_train = scaler.fit_transform(y_train)
yscaler = MinMaxScaler(feature_range=(0, 1))
y_test = yscaler.fit_transform(y_test)
```

In [18]:

```
#normalize x_train and x_test
values = x_train.values

# integer encode direction
encoder = LabelEncoder()
values[:,4] = encoder.fit_transform(values[:,4]) # ensure all data
is float
values = values.astype('float32')
# normalize features
scaler = MinMaxScaler(feature_range=(0, 1))
scaled = scaler.fit_transform(values)
x_train=pd.DataFrame(scaled)
values = x_test.values
# integer encode direction
encoder = LabelEncoder()
values[:,4] = encoder.fit_transform(values[:,4]) # ensure all data
is float
values = values.astype('float32')
# normalize features
scaler = MinMaxScaler(feature_range=(0, 1))
scaled = scaler.fit_transform(values)
x_test=pd.DataFrame(scaled)
```

The GRU unit, like traditional LSTM, require three dimensional matrixes, so here the matrixes are reshaped from 2D to 3D to fit the dimensional requirements of the neural nets.

In [19]:

```
#reshape x_train and x_test from 2D to 3D
x_train=x_train.values
x_test=x_test.values
x_train = x_train.reshape(x_train.shape[0],24,14)
x_test = x_test.reshape(x_test.shape[0],24,14)
```

I decided on a 3 convolutional layer followed by 3 GRU units then followed by one dense layer for my final neural network architecture. GRU, a time sensitive architecture, uses an update gate and reset gate. Two vectors which decided what info should be outputed. They can be trained to keep information from long ago, without diluting it through time. The update gate decides how much of past to retain, the reset gate decides how much of it to forget.

In [23]:

```
model2 = Sequential()
model2.add(Conv1D(activation='relu', input_shape=(24, 14), strides=
1, filters=15, kernel_size=20))
model2.add(Dropout(0.25))
model2.add(Conv1D(activation='relu', input_shape=(3, 8),strides=1,f
ilters=2, kernel_size=1))
model2.add(Dropout(0.25))
model2.add(Conv1D(activation='relu', input_shape=(4, 2),strides=1,
filters=2, kernel_size=1))
model2.add(Dropout(0.25))
model2.add(GRU(units=1000, input_shape=(24,14),activation='tanh',dr
opout=0.25,recurrent_dropout=0.25,return_sequences=True))
model2.add(GRU(1000, input_shape=(24,14),activation='tanh',dropout=
0.25,recurrent_dropout=0.25,return_sequences=True))
model2.add(GRU(1000,activation='tanh',dropout=0.25,recurrent_dropou
t=0.25))
model2.add(Dense(1))
```

```
#use linear function for regression because softmax is for classifi

cation
model2.add(Activation('linear'))
model2.add(Dropout(0.25))
model2.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              
Param #   
=================================================================
conv1d_1 (Conv1D)            (None, 5, 15)             
4215      
_________________________________________________________________
dropout_1 (Dropout)          (None, 5, 15)             
0         
_________________________________________________________________
conv1d_2 (Conv1D)            (None, 5, 2)              
32        
_________________________________________________________________
dropout_2 (Dropout)          (None, 5, 2)              
0         
_________________________________________________________________
conv1d_3 (Conv1D)            (None, 5, 2)              
6         
_________________________________________________________________
dropout_3 (Dropout)          (None, 5, 2)              
0         
_________________________________________________________________
```

```
_________________
gru_1 (GRU)                  (None, 5, 1000)
3009000
_________________________________________________________________
_________________
gru_2 (GRU)                  (None, 5, 1000)
6003000
_________________________________________________________________
_________________
gru_3 (GRU)                  (None, 1000)
6003000
_________________________________________________________________
_________________
dense_1 (Dense)              (None, 1)
1001
_________________________________________________________________
_________________
activation_1 (Activation)    (None, 1)
0
_________________________________________________________________
_________________
dropout_4 (Dropout)          (None, 1)
0
=================================================================
=============

Total params: 15,020,254
Trainable params: 15,020,254
Non-trainable params: 0
_________________________________________________________________
_____________
```

Compile and train the model, and assign the results to "history"

In [24]:

```
model2.compile(loss='mse', optimizer='rmsprop')
```

```
history=model2.fit(x_train, y_train, batch_size=1000,validation_dat

a=(x_test,y_test), epochs = 30)
```

```
Train on 52565 samples, validate on 17472 samples
Epoch 1/30
52565/52565 [==============================] - 26s 4
86us/step - loss: 12.3158 - val_loss: 0.0339
Epoch 2/30
52565/52565 [==============================] - 19s 3
69us/step - loss: 0.1060 - val_loss: 0.0218
Epoch 3/30
52565/52565 [==============================] - 19s 3
69us/step - loss: 0.1143 - val_loss: 0.0331
Epoch 4/30
52565/52565 [==============================] - 19s 3
71us/step - loss: 0.1273 - val_loss: 0.0261
Epoch 5/30
52565/52565 [==============================] - 20s 3
71us/step - loss: 0.1012 - val_loss: 0.2395
Epoch 6/30
52565/52565 [==============================] - 20s 3
73us/step - loss: 0.1053 - val_loss: 0.0251
Epoch 7/30
52565/52565 [==============================] - 20s 3
72us/step - loss: 0.0978 - val_loss: 0.0258
Epoch 8/30
52565/52565 [==============================] - 20s 3
73us/step - loss: 0.1003 - val_loss: 0.0292
Epoch 9/30
52565/52565 [==============================] - 20s 3
74us/step - loss: 0.0982 - val_loss: 0.0213
Epoch 10/30
52565/52565 [==============================] - 20s 3
73us/step - loss: 0.0966 - val_loss: 0.0229
Epoch 11/30
52565/52565 [==============================] - 20s 3
```

```
73us/step - loss: 0.0982 - val_loss: 0.0226
Epoch 12/30
52565/52565 [==============================] - 20s 3
74us/step - loss: 0.0959 - val_loss: 0.0086
Epoch 13/30

52565/52565 [==============================] - 20s 3
74us/step - loss: 0.0964 - val_loss: 0.0190
Epoch 14/30
52565/52565 [==============================] - 20s 3
75us/step - loss: 0.0965 - val_loss: 0.0164
Epoch 15/30
52565/52565 [==============================] - 20s 3
76us/step - loss: 0.0954 - val_loss: 0.0217
Epoch 16/30
52565/52565 [==============================] - 20s 3
74us/step - loss: 0.0954 - val_loss: 0.0419
Epoch 17/30
52565/52565 [==============================] - 20s 3
75us/step - loss: 0.0947 - val_loss: 0.0148
Epoch 18/30
52565/52565 [==============================] - 20s 3
75us/step - loss: 0.0940 - val_loss: 0.0148
Epoch 19/30
52565/52565 [==============================] - 20s 3
75us/step - loss: 0.0950 - val_loss: 0.0167
Epoch 20/30
52565/52565 [==============================] - 20s 3
74us/step - loss: 0.0947 - val_loss: 0.0200
Epoch 21/30
52565/52565 [==============================] - 20s 3
75us/step - loss: 0.0925 - val_loss: 0.0170
Epoch 22/30
52565/52565 [==============================] - 20s 3
73us/step - loss: 0.0924 - val_loss: 0.0226
Epoch 23/30
52565/52565 [==============================] - 20s 3
```

```
74us/step - loss: 0.0920 - val_loss: 0.0327
Epoch 24/30
52565/52565 [==============================] - 20s 3
76us/step - loss: 0.0950 - val_loss: 0.0271
Epoch 25/30

52565/52565 [==============================] - 20s 3
74us/step - loss: 0.0947 - val_loss: 0.0219
Epoch 26/30
52565/52565 [==============================] - 20s 3
74us/step - loss: 0.0931 - val_loss: 0.0216
Epoch 27/30
52565/52565 [==============================] - 20s 3
75us/step - loss: 0.0928 - val_loss: 0.0199
Epoch 28/30
52565/52565 [==============================] - 20s 3
73us/step - loss: 0.0918 - val_loss: 0.0173
Epoch 29/30
52565/52565 [==============================] - 20s 3
74us/step - loss: 0.0930 - val_loss: 0.0191
Epoch 30/30
52565/52565 [==============================] - 20s 3
76us/step - loss: 0.0919 - val_loss: 0.0270
```

Graphing the loss over the training epochs, it is seen that it quickly converges to around 0.02.

In [25]:

```
# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```