cccc

1.a To illustrate the plot we can consider the case of having feature x and corresponding real-value labeled y,we can say

$$y_i = \sum_{i=1}^{d} x^i w_i$$

As we have to consider that the degree of the polynomial is increasing we can say that the complexity of the model incre

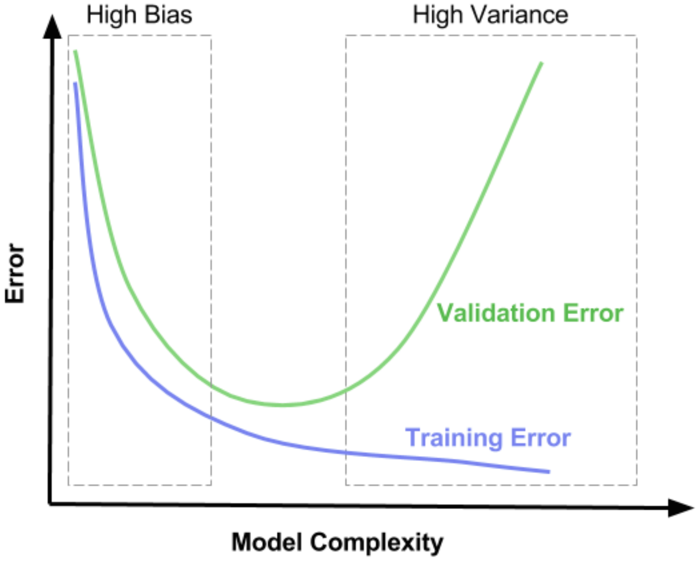


Figure 1: Plot for train and test error against Model Complexity

The x-axis represents model complexity. This has to do with how flexible our model is. Some things that add complexity to a model include: additional features, increasing polynomial terms.

The y-axis is the error in the training and the test data, it is often the Mean-Squared Error (MSE) **for regression Models.**

The blue curve is Training Error. Notice that it only decreases. It is obvious that adding model complexity leads to smaller and smaller training errors.

The green curve forms a U-shape. This curve represents Validation Error. Now as discussed in our class lectures that the test error cannot be calculated precisely but it can be estimated ,so validation error gives us the gist of how the model will fit out test data. First it decreases, hits a minimum, and then increases.

1.b *A good thumb rule from the above plot can be generalised as whenever a model has High bias and low variance it is under fitting and whenever we have low bias and high variance we have over fitting of data. The best model representation can be achieved in a number of ways, first is to find the optimal point for the bias-variance trade off which is the minimum point of the test error and it will be the best model representation.*

Using more training examples fixes high variance but not high bias. Fewer features fixes high variance but not high bias.Additional features fixes high bias but not high variance.The addition of polynomial and interaction features fixes high bias but not high variance.

Model Order Selection for Neural Data

link text (https://)**Name**: Farhan Rahman

Net ID: fr2119

Attribution: This notebook is a slightly adapted version of the model order selection lab assignment (https://github.com/sdrangan/introml/blob/master/unit04_model_sel/lab_neural_partial.ipynb) by Prof. Sundeep Rangan.

Machine learning is a key tool for neuroscientists to understand how sensory and motor signals are encoded in the brain. In addition to improving our scientific understanding of neural phenomena, understanding neural encoding is critical for brain machine interfaces. In this notebook, you will use model selection for performing some simple analysis on real neural signals.

Loading the data

The data in this lab comes from neural recordings described in:

Stevenson, Ian H., et al. "Statistical assessment of the stability of neural movement representations." Journal of neurophysiology 106.2 (2011): 764-774 (http://jn.physiology.org/content/106/2/764.short)

Neurons are the basic information processing units in the brain. Neurons communicate with one another via *spikes* or *action potentials* which are brief events where voltage in the neuron rapidly rises then falls. These spikes trigger the electro-chemical signals between one neuron and another. In this experiment, the spikes were recorded from 196 neurons in the primary motor cortex (M1) of a monkey using an electrode array implanted onto the surface of a monkey's brain. During the recording, the monkey performed several reaching tasks and the position and velocity of the hand was recorded as well.

The goal of the experiment is to try to *read the monkey's brain*: That is, predict the hand motion from the neural signals from the motor cortex.

We first load the key packages.

In [41]:

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

import pickle

from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score, mean_squared_error
from sklearn.model_selection import train_test_split, KFold
```

The full data is available on the CRCNS website http://crcns.org/data-sets/movements/dream (http://crcns.org/data-sets/movements/dream). However, the raw data files can be quite large. To make the lab easier, the Kording lab (http://kordinglab.com/) at UPenn has put together an excellent repository (https://github.com/KordingLab/Neural_Decoding) where they have created simple pre-processed versions of the data. You can download the file `example_data_s1.pickle` from the Dropbox link (https://www.dropbox.com/sh/n4924ipcfjqc0t6/AADOv9JYMUBK1tlg9P71gSSra/example_data_s1.pickle?dl=0). Alternatively, you can directly run the following command. This may take a little while to download since the file is 26 MB.

In [42]:

```
!wget 'https://www.dropbox.com/sh/n4924ipcfjqc0t6/AADOv9JYMUBK1tlg9P71gSSra/exampl
e_data_s1.pickle?dl=1' -O example_data_s1.pickle
```

```
--2020-07-10 08:42:29--  https://www.dropbox.com/sh/n4924ipcfjqc0t6/AA
DOv9JYMUBK1tlg9P71gSSra/example_data_s1.pickle?dl=1
Resolving www.dropbox.com (www.dropbox.com)... 162.125.82.1, 2620:100:
6032:1::a27d:5201
Connecting to www.dropbox.com (www.dropbox.com)|162.125.82.1|:443... c
onnected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: /sh/dl/n4924ipcfjqc0t6/AADOv9JYMUBK1tlg9P71gSSra/example_dat
a_s1.pickle [following]
--2020-07-10 08:42:29--  https://www.dropbox.com/sh/dl/n4924ipcfjqc0t
6/AADOv9JYMUBK1tlg9P71gSSra/example_data_s1.pickle
Reusing existing connection to www.dropbox.com:443.
HTTP request sent, awaiting response... 302 Found
Location: https://uc07699e6580655f9dda5c73a0c1.dl.dropboxusercontent.c
om/cd/0/get/A7OOR9E-pRl5skLPSBwgxtWfqJSgi48XU9P2Rw6ZwoyMfrXbYHJnjA2RYT
TyDjBahs4LDVADAV8ul51b_qNdLnWfYAk1h1tQLTFHNX33sgoqTg/file?dl=1# [follo
wing]
--2020-07-10 08:42:30--  https://uc07699e6580655f9dda5c73a0c1.dl.dropb
oxusercontent.com/cd/0/get/A7OOR9E-pRl5skLPSBwgxtWfqJSgi48XU9P2Rw6Zwoy
MfrXbYHJnjA2RYTTyDjBahs4LDVADAV8ul51b_qNdLnWfYAk1h1tQLTFHNX33sgoqTg/fi
le?dl=1
Resolving uc07699e6580655f9dda5c73a0c1.dl.dropboxusercontent.com (uc07
699e6580655f9dda5c73a0c1.dl.dropboxusercontent.com)... 162.125.82.15,
2620:100:6032:15::a27d:520f
Connecting to uc07699e6580655f9dda5c73a0c1.dl.dropboxusercontent.com
(uc07699e6580655f9dda5c73a0c1.dl.dropboxusercontent.com)|162.125.82.15
|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 26498656 (25M) [application/binary]
Saving to: ‘example_data_s1.pickle’

example_data_s1.pic 100%[===================>]  25.27M  14.4MB/s    in
1.8s

2020-07-10 08:42:32 (14.4 MB/s) - ‘example_data_s1.pickle’ saved [2649
8656/26498656]
```

The file is a *pickle* data structure, which uses the Python package `pickle` to serialize Python objects into data files. Once you have downloaded the file, you can run the following command to retrieve the data from the pickle file.

In [43]:

```
with open('example_data_s1.pickle', 'rb') as fp:
    X,y = pickle.load(fp)
```

The matrix `X` is matrix of spike counts from different neurons, where `X[i,j]` is the number of spikes from neuron `j` in time bin `i`.

The matrix `y` has two columns:

- `y[i,0]` = velocity of the monkey's hand in the x-direction
- `y[i,1]` = velocity of the monkey's hand in the y-direction Our goal will be to predict `y` from `X`.

Each time bin represent `tsamp=0.05` seconds of time. Using `X.shape` and `y.shape`, we can compute and print:

- `nt` = the total number of time bins
- `nneuron` = the total number of neurons
- `nout` = the total number of output variables to track = number of columns in `y`
- `ttotal` = total time of the experiment is seconds.

In [44]:

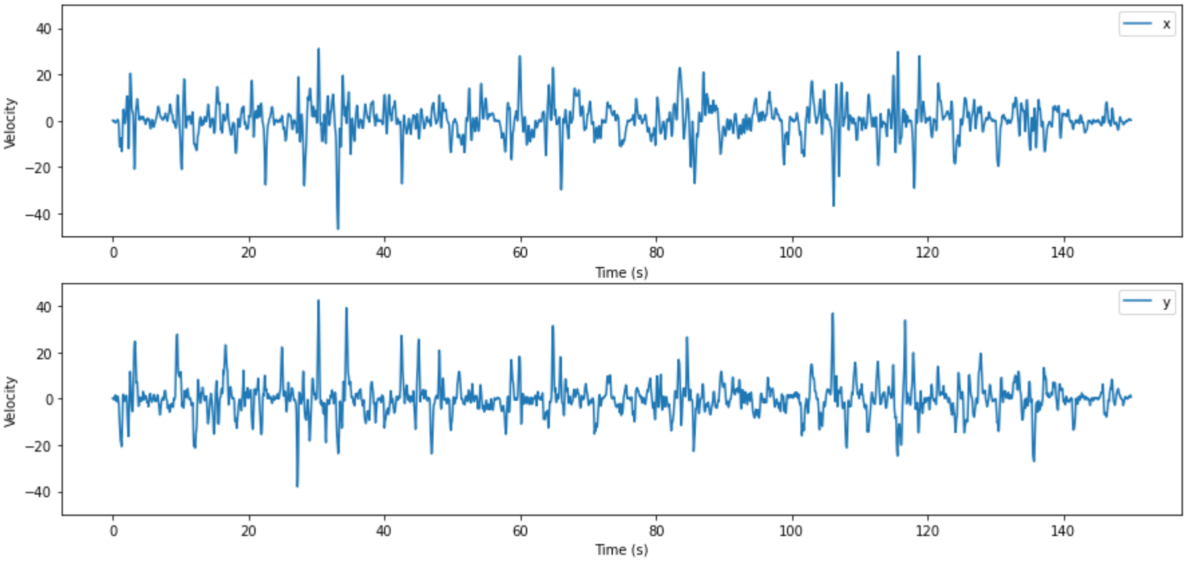
```
tsamp = 0.05  # sampling time in seconds

nt, nneuron = X.shape
nout = y.shape[1]
ttotal = nt*tsamp

print('Number of neurons = %d' % nneuron)
print('Number of time samples = %d' % nt)
print('Number of outputs = %d' % nout)
print('Total time (secs) = %f' % ttotal)
```

```
Number of neurons = 52
Number of time samples = 61339
Number of outputs = 2
Total time (secs) = 3066.950000
```

Then, we can plot the velocity against time, for each direction, for the first few minutes:

In [45]:

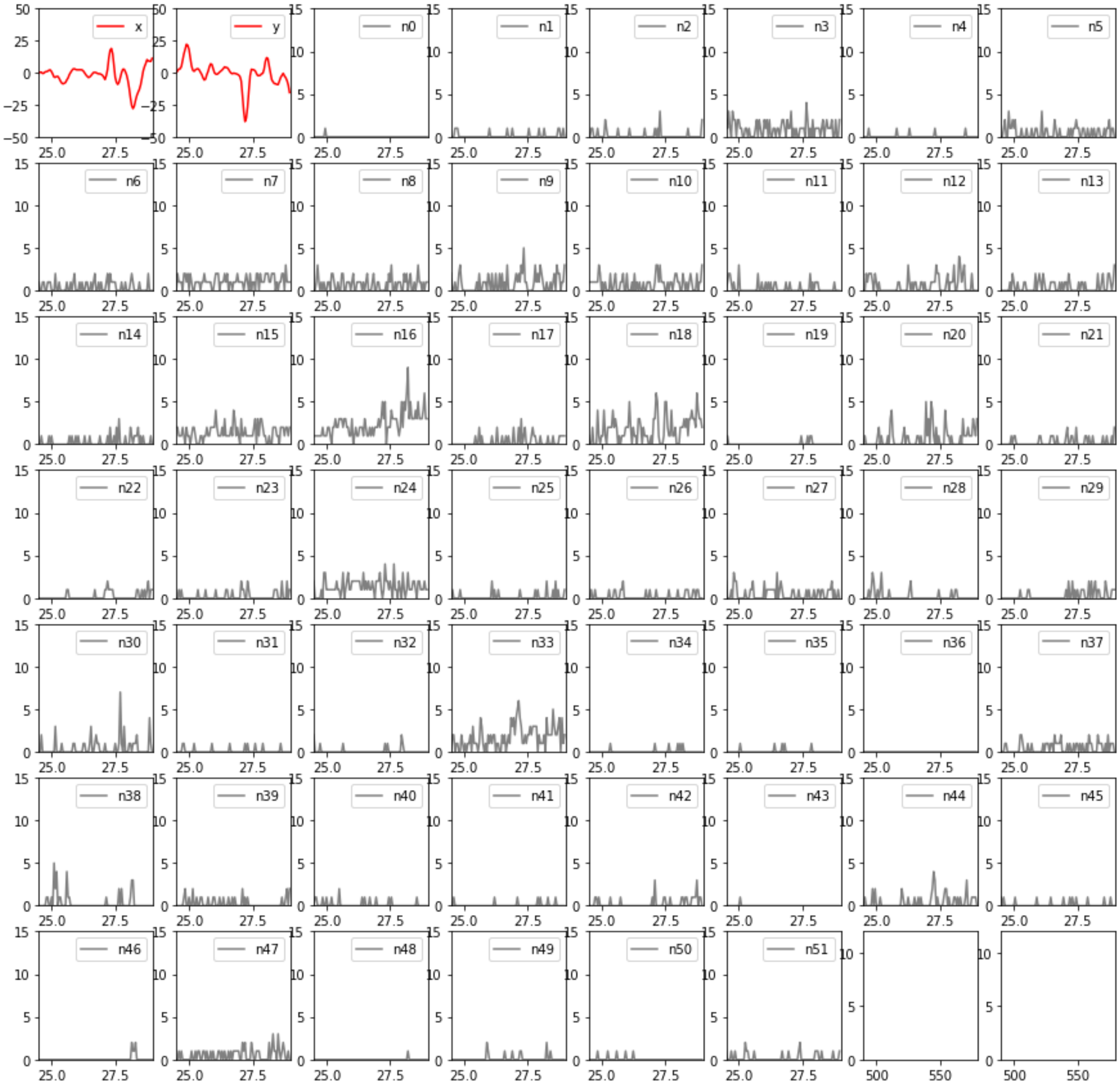
```
t_cutoff = 3000
directions = ['x', 'y']

fig, axes = plt.subplots(nrows=2, ncols=1, figsize=(15,7))
for n in range(nout):
  sns.lineplot(np.arange(0, t_cutoff)*tsamp, y[0:t_cutoff, n], label=directions[n
], ax=axes[n]);

  axes[n].set_ylabel("Velocity")
  axes[n].set_xlabel("Time (s)")
  axes[n].set_ylim(-50,50)
```



Then, we can "zoom in" on a small slice of time in which the monkey is moving the hand, and see the neural activity at the same time.

In [46]:

```
t_start = 490
t_end = 580

fig, axes = plt.subplots(nrows=7, ncols=8, figsize=(15,15))

# Setting the range for all axes
plt.setp(axes, xlim=(t_start, t_end), ylim=(0,12));

for n in range(nout):
  sns.lineplot(np.arange(t_start, t_end)*tsamp, y[t_start:t_end, n], ax=axes[n//2, n%2], color='red', label=directions[n])
  plt.setp(axes[n//2,n%2], xlim=(t_start*tsamp, t_end*tsamp), ylim=(-50, +50));

for n in range(nneuron):
  sns.lineplot(np.arange(t_start, t_end)*tsamp, X[t_start:t_end, n], ax=axes[(n+2)//8,(n+2)%8], label="n%d" % n, color='grey')
  plt.setp(axes[(n+2)//8,(n+2)%8], xlim=(t_start*tsamp, t_end*tsamp), ylim=(0, +15));
```



Fitting a simple linear model

Let's first try a simple linear regression model to fit the data.

To start, we will split the data into a training set and a test set. We'll fit the model on the training set and then use the test set to estimate the model performance on new, unseen data.

To shuffle or not to shuffle?

The `train_test_split` function has an optional `shuffle` argument.

- If you use `shuffle=False`, then `train_test_split` will take the first part of the data as the training set and the second part of the data as the test set, according to the ratio you specify in `test_size` or `train_size`.
- If you use `shuffle=True`, then `train_test_split` will first randomly shuffle the data. Then, it will take the first part of the *shuffled* data as the training set and the second part of the *shuffled* data as the test set, according to the ratio you specify in `test_size` or `train_size`.

According to the function [documentation (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html)](https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html), by default, `shuffle` is `True`:

> **shuffle: bool, default=True**
>
> Whether or not to shuffle the data before splitting. If shuffle=False then stratify must be None.

so if you do not specify anything related to `shuffle`, your data will be randomly shuffled before it is split into training and test data.

Under what conditions should you shuffle data? Suppose your dataset includes samples of a medical experiment on 1000 subjects, and the first 500 samples in the data are from male subjects while the second 500 samples are from female subjects. If you set `shuffle=False`, then your training set would have a much higher proportion of male subjects than your test set (with the specific numbers depending on the ratio you specify).

On the other hand, suppose your dataset includes stock prices at closing time, with each sample representing a different date (in order). If you allow `train_test_split` to shuffle the data, then your model will be allowed to "learn" stock prices using prices from the day *after* the one it is trying to predict! Obviously, your model won't be able to learn from future dates in production, so it shouldn't be allowed to in the evaluation stage, either. (Predicting the past using the future is considered a type of data leakage.)

With this in mind, it is usually inappropriate to shuffle time series data when splitting it up into smaller sets for training, validation, or testing.

(There are more sophisticated ways to handle splitting time series data, but for now, splitting it up the usual way, just without shuffling first, will suffice.)

Given the discussion above, use the `train_test_split` function to split the data into training and test sets, but with no shuffling. Let `Xtr,ytr` be the training data set and `Xts,yts` be the test data set. Use `test_size=0.33` so 1/3 of the data is used for evaluating the model performance.

In [47]:

```
# TODO 1
from sklearn.model_selection import train_test_split

# TODO
Xtr, Xts, ytr, yts= train_test_split(X, y, test_size=0.33,shuffle=False)
```

Now, fit a linear regression on the training data `Xtr,ytr` . Make a prediction `yhat` using the test data, `Xts` . Compare `yhat` to `yts` to measure `rsq` , the R2 value. You can use the `r2_score` method. Print the `rsq` value. You should get `rsq` of around `0.45` .

In [48]:

```
# TODO 2
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score

reg = LinearRegression()
reg.fit(Xtr,ytr)
reg.coef_

yhat = reg.predict(Xts)
rsq= r2_score(yts, yhat)
print(rsq)
```

```
0.4499831346553009
```

It is useful to plot the predicted vs. actual values. Since we have two predicted values for each sample - the velocity in the X direction and the velocity in the Y direction - you should make two subplots,

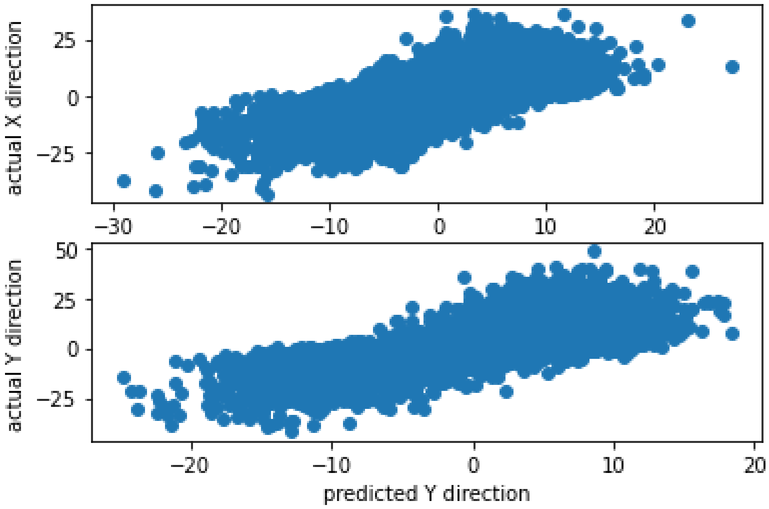
- one of predicted X direction vs. actual X direction,
- one of predicted Y direction vs. actual Y direction

In [49]:

```
# TODO 3

plt.subplot(2,1,1)
plt.scatter(yhat[:,0],yts[:,0])
plt.xlabel('predicted X direction')
plt.ylabel('actual X direction')
#plt.figure(figsize=(20,10))

plt.subplot(2,1,2)
plt.scatter(yhat[:,1],yts[:,1])
plt.xlabel('predicted Y direction')
plt.ylabel('actual Y direction')
#plt.figure(figsize=(20,10))
```

Out[49]:

```
Text(0, 0.5, 'actual Y direction')
```



Fitting a model with delay

One way we can improve the model accuracy is to add features using delayed version of the existing features.

Specifically, the model we used above tries to predict velocity in direction $k$ at time $i$ using $$\hat{y}_{i,k} = w_{k,0} + \sum_{d=1}^{\text{nneuron}} w_{k,d} X_{i,d} $$

In this model, $\hat{y}_{i,k}$ at the $i$th time bin was only dependent on $X_i$, the number of spikes of each neuron in time bin $i$. In signal processing, this is called a *memoryless* model.

However, in many physical systems, such as those that arise in neuroscience, there is a delay between the inputs and outputs. To model this effect, we could add additional features to each row of data, representing the number of spikes of each neuron in the *previous* row. Then, the output at time $i$ would be modeled as the effect of the neurons firing in time $i$ *and* the effect of the neurons firing in time $i-1$.

We wouldn't be able to use data from the past for the first row of data, since we don't *have* data about neurons firing in the previous time step. But we can drop that row. If our original data matrix had `nt` rows and `nneuron` columns, our data matrix with delayed features would have `nt - 1` rows and `nneuron + 1 x nneuron` columns. (The first `nneuron` columns represent the number of spikes in each neuron for the current time, the next `nneuron` columns represent the number of spikes in each neuron for the previous time.)

Furthermore, we can "look back" any number of time steps, so that the output at time $i$ is modeled as the effect of the neurons firing in time $i$, the neurons firing in time $i-1$, ..., all the way up to the effect of the neurons firing in time $i- \text{dly}$ (where $\text{dly}$ is the maximum number of time steps we're going to "look back" on). Our data matrix with the additional delayed features would have `nt - dly` rows and `nneuron + dly x nneuron` columns.

Here is a function that accepts `X` and `y` data and a `dly` argument, and returns `X` and `y` with delayed features up to `dly` time steps backward.

In [50]:

```
def create_dly_data(X,y,dly):
    """
    Create delayed data
    """
    n,p = X.shape
    Xdly = np.zeros((n-dly,(dly+1)*p))
    for i in range(dly+1):
        Xdly[:,i*p:(i+1)*p] = X[dly-i:n-i,:]
    ydly = y[dly:]

    return Xdly, ydly
```

To convince yourself that this works, try creating a data matrix that includes delayed features one time step back:

In [51]:

```
X_dly1, y_dly1 = create_dly_data(X, y, 1)
```

Verify that the dimensions have changed, as expected:

In [52]:

```
# dimensions of original data matrix
X.shape
```

Out[52]:

```
(61339, 52)
```

In [53]:

```
# dimensions of data matrix with delayed features 1 time step back
X_dly1.shape
```

Out[53]:

```
(61338, 104)
```

Check row 0 in the matrix with delayed features, and verify that it is the concatenation of row 1 and row 0 in the original data matrix. (Note that row 0 in the matrix with delayed features corresponds to row 1 in the original data matrix.)

In [54]:

```
X_dly1[0]
```

Out[54]:

```
array([0., 0., 1., 1., 0., 1., 0., 0., 1., 0., 0., 0., 1., 0., 0., 0.,
0.,
       0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 3., 0., 0., 0., 0.,
0.,
       0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0.,
0.,
       1., 0., 0., 0., 1., 2., 0., 0., 1., 0., 2., 0., 0., 3., 0., 0.,
2.,
       2., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 1., 0., 2., 0.,
0.,
       2., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 1., 0., 1., 0.,
0.,
       0., 1.])
```

In [55]:

```
np.hstack((X[1], X[0]))
```

Out[55]:

```
array([0., 0., 1., 1., 0., 1., 0., 0., 1., 0., 0., 0., 1., 0., 0., 0.,
0.,
       0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 3., 0., 0., 0., 0.,
0.,
       0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0.,
0.,
       1., 0., 0., 0., 1., 2., 0., 0., 1., 0., 2., 0., 0., 3., 0., 0.,
2.,
       2., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 1., 0., 2., 0.,
0.,
       2., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 1., 0., 1., 0.,
0.,
       0., 1.])
```

In [56]:

```
y_dly1[0]
```

Out[56]:

```
array([-0.13949835,  0.11006426])
```

In [57]:

```
y[1]
```

Out[57]:

```
array([-0.13949835,  0.11006426])
```

Now fit an linear delayed model with `dly=2` delay lags. That is,

- Create delayed data `Xdly,ydly=create_dly_data(X,y,dly=2)`
- Split the data into training and test as before
- Fit the model on the training data
- Measure the `R^2` score on the test data

If you did this correctly, you should get a new `R^2` score around 0.60. This is significantly better than the memoryless models.

In [58]:

```
# TODO 4

# Create the delayed data
Xdly,ydly=create_dly_data(X,y,dly=2)

# Split into training and test

Xdtr, Xdts, ydtr, ydts= train_test_split(Xdly, ydly, test_size=0.33,shuffle=False)

# Create linear regression object

reg_d = LinearRegression()
# Fit the model

reg_d.fit(Xdtr,ydtr)

# Measure the new r2 score
yd_hat = reg_d.predict(Xdts)
rsq2= r2_score(ydts, yd_hat)
print(rsq2)
```

```
0.6033897697058304
```

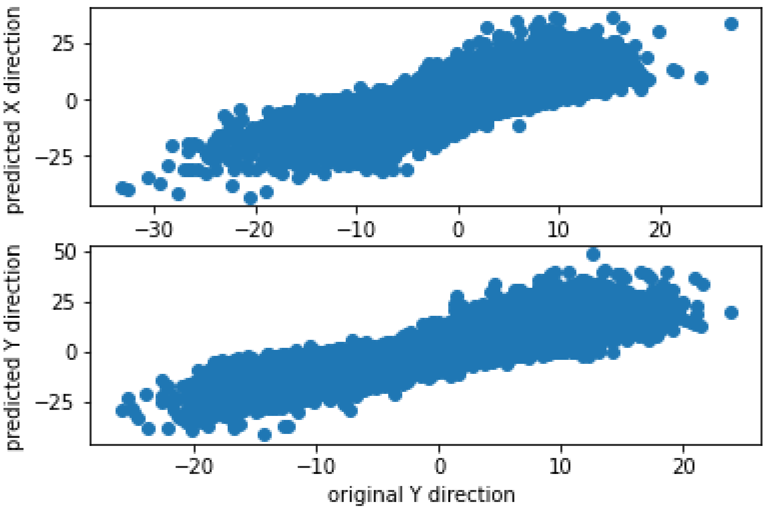
Plot the predicted vs. true values as before, with one subplot for. You should visually see a better fit.

In [59]:

```
# TODO 5

plt.subplot(211)
plt.scatter(yd_hat[:, 0], ydts[:, 0])
plt.ylabel('predicted X direction')
plt.xlabel('original X direction')
plt.subplot(212)
plt.scatter(yd_hat[:, 1], ydts[:, 1])
plt.ylabel('predicted Y direction')
plt.xlabel('original Y direction ')
```

Out[59]:

```
Text(0.5, 0, 'original Y direction ')
```



Selecting the optimal delay with K-fold CV

In the previous example, we fixed `dly=2` . We can now select the optimal delay using K-fold cross validation.

Since we have a large number of data samples, it turns out that the optimal model order uses a very high delay. Using the above fitting method, the computations take too long. So, to simplify things, we will first just pretent that we have a very limited data set.

We will compute `Xred` and `yred` by taking the first `nred=6000` samples of the data `X` and `y` . This is about 10% of the overall data.

In [60]:

```
nred = 6000

Xred = X[:nred]
yred = y[:nred]
```

We will look at model orders up to `dmax=15` . We will create a delayed data matrix, `Xdly,ydly` , using `create_dly_data` with the reduced data `Xred,yred` and `dly=dmax` .

In [61]:

```
dmax = 15

Xdly, ydly = create_dly_data(Xred,yred,dmax)
```

In [62]:

```
Xdly.shape
```

Out[62]:

```
(5985, 832)
```

In [63]:

```
ydly.shape
```

Out[63]:

```
(5985, 2)
```

Note that we can use `Xdly, ydly` to get a data matrix for any delay *up to* `dmax` , not only for delay = `dmax` . For example, to get a data matrix with delay = 1:

In [64]:

```
dtest = 1
X_dtest = Xdly[:,:(dtest+1)*nneuron]
X_dtest.shape
```

Out[64]:

```
(5985, 104)
```

We are going to use K-fold CV with `nfold=10` to find the optimal delay, for all the values of delay in `dtest_list` :

In [65]:

```
dtest_list = np.arange(0, dmax+1)
nd = len(dtest_list)

print(dtest_list)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
```

You can refer to the example in the "Model order selection" section of the demo notebook, but you will have to make some changes:

- Make sure to use `shuffle=False` in your `KFold` object, since you are working with time series data.
- In the demo notebook, we recorded MSE on the validation set. Here, we'll use R2 instead.
- In the demo notebook, when finding the "best" model order, we were using a "lower is better" metric. Here, you'll follow the procedure for a "higher is better" metric, since you are using R2.

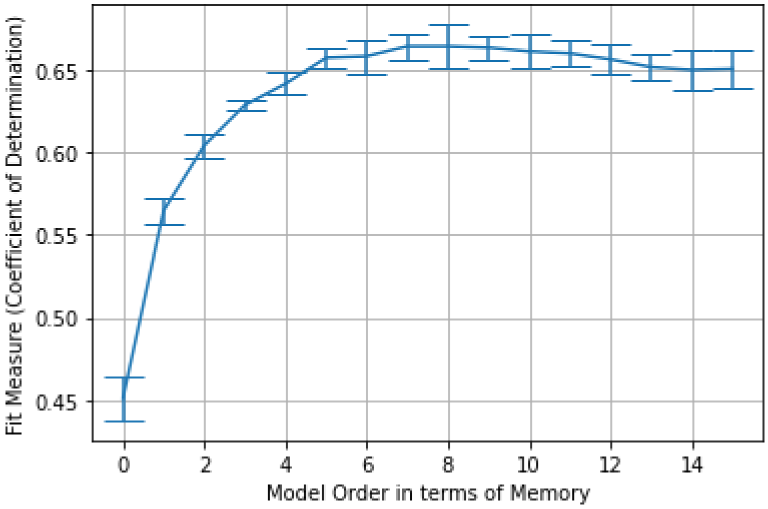
In [66]:

```
import  sklearn.model_selection
# Number of folds
nfold = 10

# TODO 6  Create a k-fold object
# kf = KFold(...)
kf = sklearn.model_selection.KFold(n_splits=nfold,shuffle=False)


# TODO 7
# Initialize a matrix Rsq to hold values of the R^2 across the model orders and fo
lds.
# Rsq = ...
Rsq = np.zeros((nfold,nd))

# Loop over the folds

for isplit, idx in enumerate(kf.split(Xdly)):

    # Get the training and validation data in the split
    idx_tr, idx_val = idx

    for it, dtest in enumerate(dtest_list):
        # TODO 8
        # don't call create_dly_data again
        # just select the appropriate subset of columns of Xdly
        # X_dtest = Xdly with the columns corresponding to only the `dtest+1` most
recent times.

        # TODO 9
        # Split the data (X_dtest,ydly) into training and validation
        # using idx_tr and idx_val
        # Xtr = ...
        # ytr = ...
        # Xval = ...
        # yval = ...
        X_dl = Xdly[:,:(dtest+1)*X.shape[1]]
        Xd_ytr, Xd_yts, yd_ytr, yd_yts = train_test_split(X_dl, ydly, test_size=0.
33)



        # TODO 10  Fit linear regression on training data
        regr = LinearRegression()
        regr.fit(Xd_ytr, yd_ytr)

        yd_yhat = regr.predict(Xd_yts)
        # TODO 11  Measure the R2 on validation data and store in the matrix Rsq
        rd_ysq = r2_score(yd_yts, yd_yhat)
        Rsq[isplit, dtest] = rd_ysq
```

Compute the mean and standard error of the R2 values for each model (each delay value) and plot it as a function of the delay. Use a `plt.errorbar` plot, as shown in the "Model selection using 1-SE rule" section of the demo notebook. Label your axes.

In [67]:

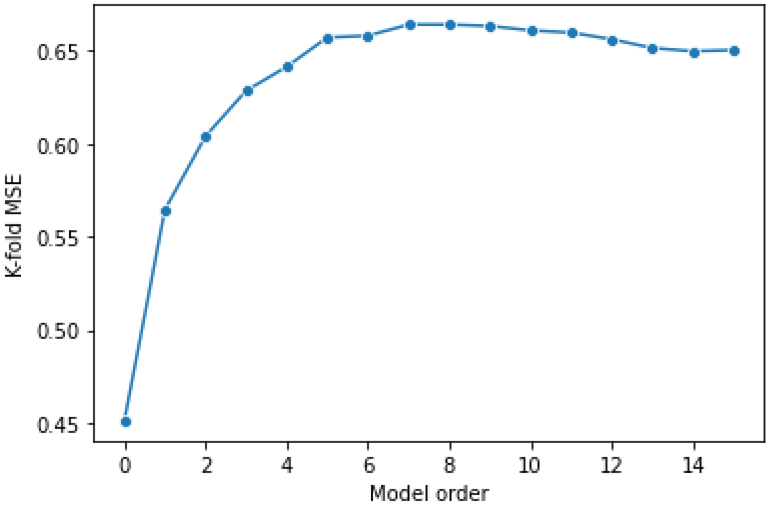
```
# TODO 12
Rsqstd = np.std(Rsq, axis=0)
Rsq_mean = np.mean(Rsq,axis=0)
plt.figure()
plt.errorbar(dtest_list,Rsq_mean,yerr=Rsqstd,capsize=10)

plt.ylabel('Fit Measure (Coefficient of Determination)')
plt.xlabel('Model Order in terms of Memory')
plt.grid()
```



Write code to find the delay that has the highest validation R2. Print the best delay according to the "highest R2" rule.

In [68]:

```
sns.lineplot(x=dtest_list, y=Rsq.mean(axis=0), marker='o');
plt.xlabel("Model order");
plt.ylabel("K-fold MSE");
```



In [69]:

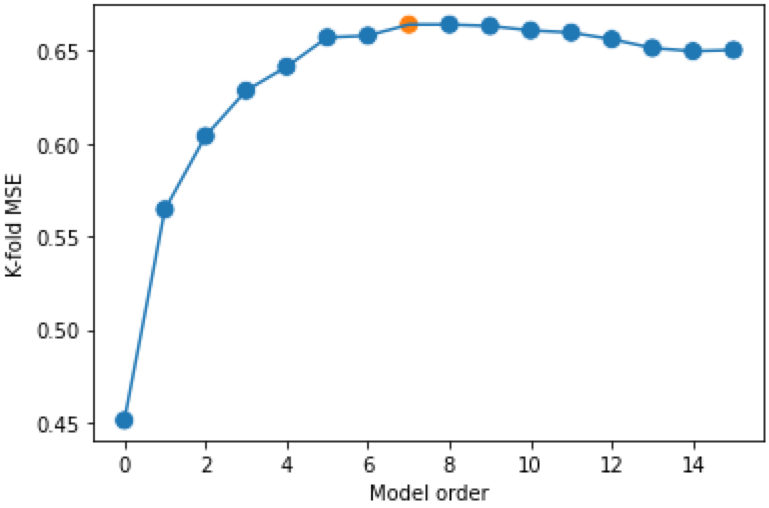
```
idx_max = np.argmax(Rsq.mean(axis=0))
d_max_mse = dtest_list[idx_max]
d_max_mse
```

Out[69]:

```
7
```

In [70]:

```
sns.lineplot(x=dtest_list, y=Rsq.mean(axis=0));
sns.scatterplot(x=dtest_list, y=Rsq.mean(axis=0), hue=dtest_list==d_max_mse, s=100
, legend=False);

plt.xlabel("Model order");
plt.ylabel("K-fold MSE");
```



Now write code to find the best delay using the one SE rule (i.e. find the simplest model whose validation R2 is within one SE of the model with the highest R2). Print the best delay according to the "one SE rule."

In [71]:

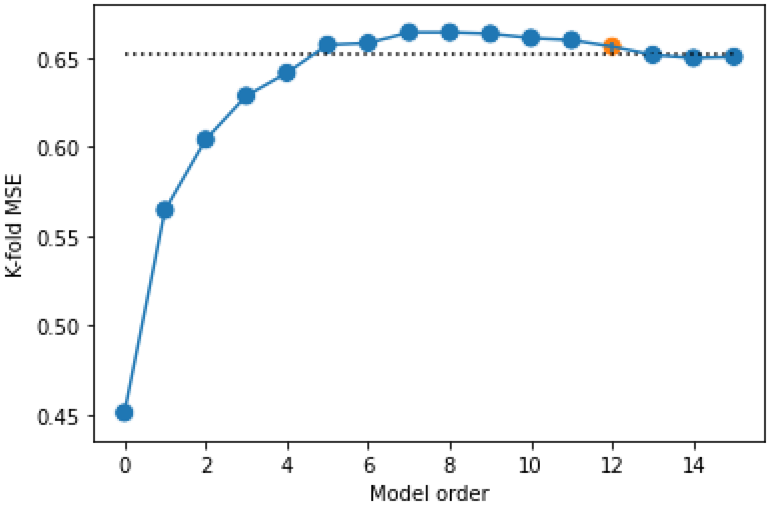
```
idx_max = np.argmax(Rsq.mean(axis=0))
target = Rsq[idx_max,:].mean() + Rsq[idx_max,:].std()/np.sqrt(nfold-1)
# np.where returns indices of values where condition is satisfied
idx_one_se = np.where(Rsq.mean(axis=0) > target)
d_one_se = np.max(dtest_list[idx_one_se])
d_one_se
```

Out[71]:

```
12
```

In [72]:

```
plt.errorbar(x=dtest_list, y=Rsq.mean(axis=0), yerr=Rsq.std(axis=0)/np.sqrt(nfold-
1));
plt.hlines(y=target, xmin=np.min(dtest_list), xmax=np.max(dtest_list), ls='dotted'
)
sns.scatterplot(x=dtest_list, y=Rsq.mean(axis=0), hue=dtest_list==d_one_se, s=100,
legend=False);

plt.xlabel("Model order");
plt.ylabel("K-fold MSE");
```


K-fold MSE
Model order