

Lab: Model Selection for Neural Data

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 lab, you will use model selection for performing some simple analysis on real neural signals.

Before doing this lab, you should review the ideas in the [polynomial model selection demo](#) ([./polyfit.ipynb](#)). In addition to the concepts in that demo, you will learn to:

- Load MATLAB data
- Formulate models of different complexities using heuristic model selection
- Fit a linear model for the different model orders
- Select the optimal model via cross-validation

The last stage of the lab uses LASSO estimation for model selection. If you are doing this part of the lab, you should review the concepts in [LASSO demonstration](#) ([./prostate.ipynb](#)) on the prostate cancer dataset.

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 basic packages.

In [1]:

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
```

The full data is available on the CRCNS website <http://crcns.org/data-sets/movements/dream> (<http://crcns.org/data-sets/movements/dream>). This website has a large number of great datasets and can be used for projects as well. To make this lab easier, I have pre-processed the data slightly and placed it in the file StevensonV2.mat, which is a MATLAB file. You will need to have this file downloaded in the directory you are working on.

Since MATLAB is widely-used, python provides method for loading MATLAB mat files. We can use these commands to load the data as follows.

In [2]:

```
import scipy.io  
mat_dict = scipy.io.loadmat('StevensonV2.mat')
```

The returned structure, `mat_dict`, is a dictionary with each of the MATLAB variables that were saved in the .mat file. Use the `.keys()` method to list all the variables.

In [3]:

```
#TODO  
  
mat_dict.keys()
```

Out[3]:

```
dict_keys(['__header__', '__version__', '__globals__', 'Publication',  
'timeBase', 'spikes', 'time', 'handVel', 'handPos', 'target', 'startB  
ins', 'targets', 'startBinned'])
```

We extract two variables, `spikes` and `handVel`, from the dictionary `mat_dict`, which represent the recorded spikes per neuron and the hand velocity. We take the transpose of the `spikes` data so that it is in the form time bins \times number of neurons. For the `handVel` data, we take the first component which is the motion in the x -direction.

In [4]:

```
X0 = mat_dict['spikes'].T  
y0 = mat_dict['handVel'][0,:]  
print(X0.shape)  
y0.shape  
  
(15536, 196)
```

Out[4]:

```
(15536,)
```

The `spikes` matrix will be a `nt` \times `nneuron` matrix where `nt` is the number of time bins and `nneuron` is the number of neurons. Each entry `spikes[k,j]` is the number of spikes in time bin `k` from neuron `j`. Use the `shape` method to find `nt` and `nneuron` and print the values.

In [5]:

```
# TODO
nt , nneuron = (X0).shape
print (nt, ' * ',nneuron)
```

15536 * 196

Now extract the time variable from the mat_dict dictionary. Reshape this to a 1D array with nt components. Each entry time[k] is the starting time of the time bin k. Find the sampling time tsamp which is the time between measurements, and ttotal which is the total duration of the recording.

In [6]:

```
# TODO

time = (mat_dict['time']).reshape(-1)
print(time.shape)
print(time)
```

```
(15536,)
[ 12.591  12.641  12.691 ...,  789.241  789.291  789.341]
```

Linear fitting on all the neurons

First divide the data into training and test with approximately half the samples in each. Let Xtr and ytr denote the training data and Xts and yts denote the test data.

In [7]:

```
# TODO
SplitN= int(15536/2-1)
print(SplitN)
#print(int(3.2/1.2))

Xtr = X0[:SplitN]
ytr = y0[:SplitN]

print(np.max(y0))
Xts = X0[SplitN+1:]
yts = y0[SplitN+1:]

t1 = time[:SplitN]
t2 = time[SplitN+1:]
print(t1.shape)
print(yts.shape,Xts.shape,t2.shape)
```

```
7767
0.326468154144
(7767,)
(7768,) (7768, 196) (7768,)
```

Now, we begin by trying to fit a simple linear model using *all* the neurons as predictors. To this end, use the `sklearn.linear_model` package to create a regression object, and fit the linear model to the training data.

In [8]:

```
import sklearn.linear_model

# TODO

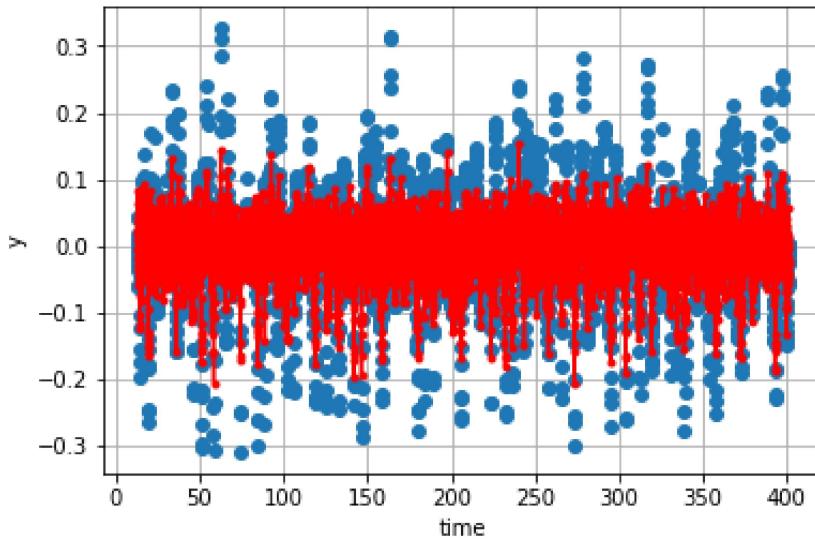
regr = sklearn.linear_model.LinearRegression()
regr.fit(Xtr,ytr)
#print(regr.coef_)

y_tr_pred = regr.predict(Xtr)

plt.plot(t1,ytr,'o',t1,y_tr_pred,'r.-')
plt.xlabel('time')
plt.ylabel('y')
plt.grid(True)

RSS_tr = np.mean((y_tr_pred-ytr)**2)/(np.std(ytr)**2)
Rsq_tr = 1-RSS_tr
print("for training :")
print("RSS for training per sample = {0:f}".format(RSS_tr))
print("R^2 =           {0:f}".format(Rsq_tr))

for training :
RSS for training per sample = 0.468816
R^2 =           0.531184
```



Measure and print the normalized RSS on the test data. Red is the `y_tr_predict`

In [9]:

```
# TODO

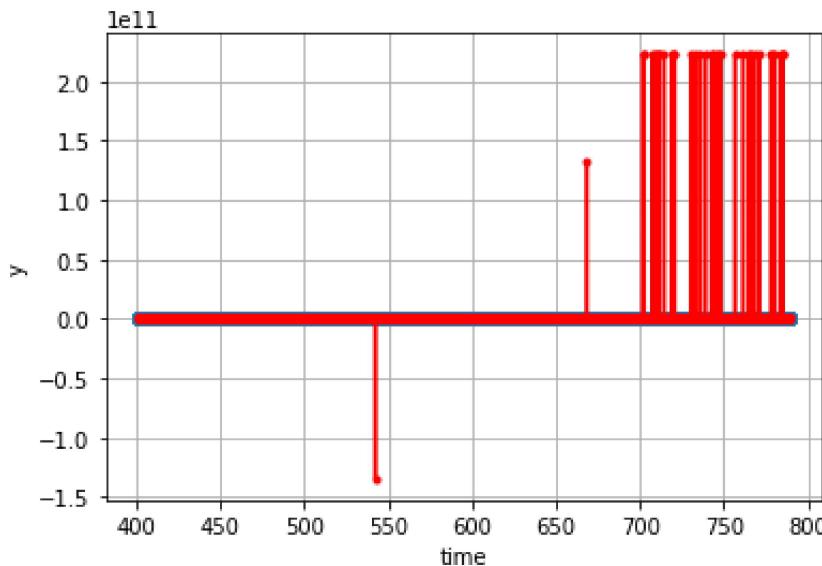
regr.fit(Xtr,ytr)
y_ts_pred = regr.predict(Xts)

plt.plot(t2,yts,'o')
plt.plot(t2,y_ts_pred,'r.-')

plt.xlabel('time')
plt.ylabel('y')
plt.grid(True)

RSS_tr = np.mean((y_ts_pred-yts)**2)/(np.std(yts)**2)
Rsq_tr = 1-RSS_tr
print("for testing :")
print("RSS for training per sample = {0:f}".format(RSS_tr))
print("R^2 = {0:f}".format(Rsq_tr))
```

```
for testing :
RSS for training per sample = 65842302859562740350976.000000
R^2 = -65842302859562740350976.000000
```



You should see that the test error is enormous -- the model does not generalize to the test data at all.

Linear Fitting with Heuristic Model Selection

The above shows that we need a way to reduce the model complexity. One simple idea is to select only the neurons that individually have a high correlation with the output.

Write code which computes the coefficient of determination, R_k^2 , for each neuron k . Plot the R_k^2 values.

You can use a for loop over each neuron, but if you want to make efficient code try to avoid the for loop and use [python broadcasting \(../Basics/numpy_axes_broadcasting.ipynb\)](#).

In [10]:

```
# TODO

Rsq = ( ((y0 - np.mean(y0)).T).dot(X0 - np.mean(X0, axis=0)) ** 2 )

#print(np.sum((y0 - np.mean(y0)) ** 2))
#print(((X0 - np.mean(X0, axis=0)) ** 2).shape)
#print(np.sum((X0 - np.mean(X0, axis=0)) ** 2, axis=0))
#print((np.sum((X0 - np.mean(X0, axis=0)) ** 2, axis=0)).shape)

div1 = (np.sum((y0 - np.mean(y0)) ** 2) * (np.sum((X0 - np.mean(X0, axis=0)) ** 2, axis=0))) / (np.sum((X0 - np.mean(X0, axis=0)) ** 2, axis=0) + 0.000001)

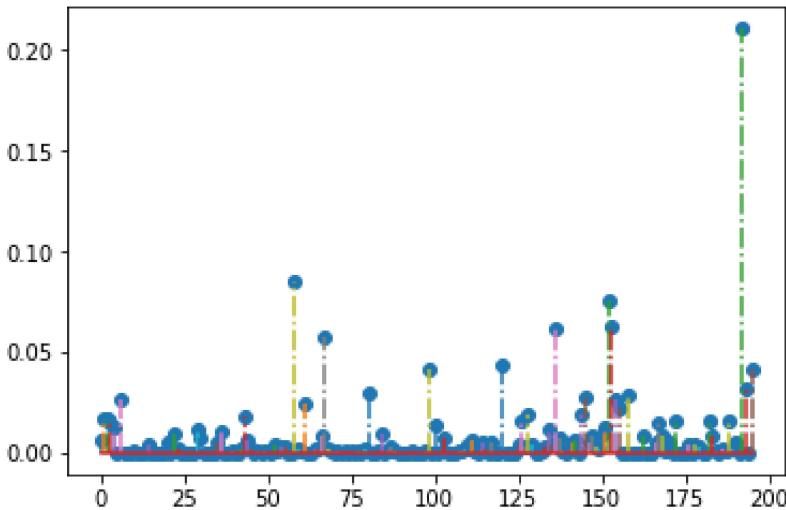
Rsq = np.true_divide(Rsq, div1)
print(Rsq.shape)
#print(Rsq)

# plt.stem...
plt.stem(range(nneuron), Rsq, '-.')
```

(196,)

Out[10]:

<Container object of 3 artists>



We see that many neurons have low correlation and can probably be discarded from the model.

Use the `np.argsort()` command to find the indices of the $d=100$ neurons with the highest R_k^2 value. Put the d indices into an array `Ise1`. Print the indices of the neurons with the 10 highest correlations.

In [11]:

```
d = 100 # Number of neurons to use

# TODO
Isel = np.argsort(Rsq)[::-1][:d]
Inotsel = np.argsort(Rsq)[::-1][d:]
print(Isel.shape, Inotsel.shape)

print("The neurons with the ten highest R^2 values = %s" % (Isel[0:10]) )

(100,) (96,)
The neurons with the ten highest R^2 values = [192 58 152 153 136 6
7 120 195 98 193]
```

Fit a model using only the d neurons selected in the previous step and print both the test RSS per sample and the normalized test RSS.

In [12]:

```
# TODO

mat_dict = scipy.io.loadmat('StevensonV2.mat')
X0 = mat_dict['spikes'].T
y0 = mat_dict['handVel'][0,:]

print(SplitN,X0.shape,y0.shape,Isel.shape)

t1 = time[:SplitN]
t2 = time[SplitN+1:]

Xtr = X0[:SplitN,Isel]
ytr = y0[:SplitN]
Xts = X0[SplitN+1:,Isel]
yts = y0[SplitN+1:]

print(Xtr.shape,ytr.shape)
print(yts.shape,Xts.shape)

regr = sklearn.linear_model.LinearRegression()
regr.fit(Xtr,ytr)
#print(regr.coef_)

# train set
y_tr_pred = regr.predict(Xtr)
#print(y_tr_pred - ytr)

plt.subplot(211)
plt.plot(t1,ytr,'bo',t1,y_tr_pred,'y.')
plt.xlabel('neuron')
plt.ylabel('y_train')
plt.grid(True)

RSS_tr = np.mean((y_tr_pred-ytr)**2)/(np.std(ytr)**2)
Rsq_tr = 1-RSS_tr
print("for training :")
print("RSS for training per sample = {0:f}".format(RSS_tr))
print("R^2 = {0:f}".format(Rsq_tr))

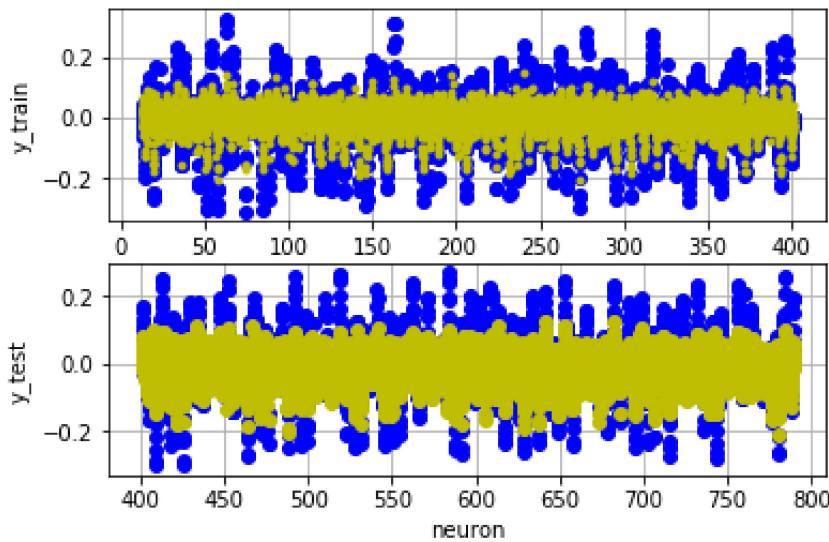
# test set

y_ts_pred = regr.predict(Xts)
#print(regr.score(Xts,y_ts_pred))

plt.subplot(212)
plt.plot(t2,yts,'bo')
plt.plot(t2,y_ts_pred,'yo')
plt.xlabel('neuron')
plt.ylabel('y_test')
plt.grid(True)
```

```
RSS_tr = np.mean((y_ts_pred-yts)**2)/(np.std(yts)**2)
Rsq_tr = 1-RSS_tr
print("for testing :")
print("RSS for training per sample = {0:f}".format(RSS_tr))
print("R^2 = {0:f}".format(Rsq_tr))
```

```
7767 (15536, 196) (15536,) (100,)
(7767, 100) (7767,)
(7768,) (7768, 100)
for training :
RSS for training per sample = 0.480873
R^2 = 0.519127
for testing :
RSS for training per sample = 0.496146
R^2 = 0.503854
```

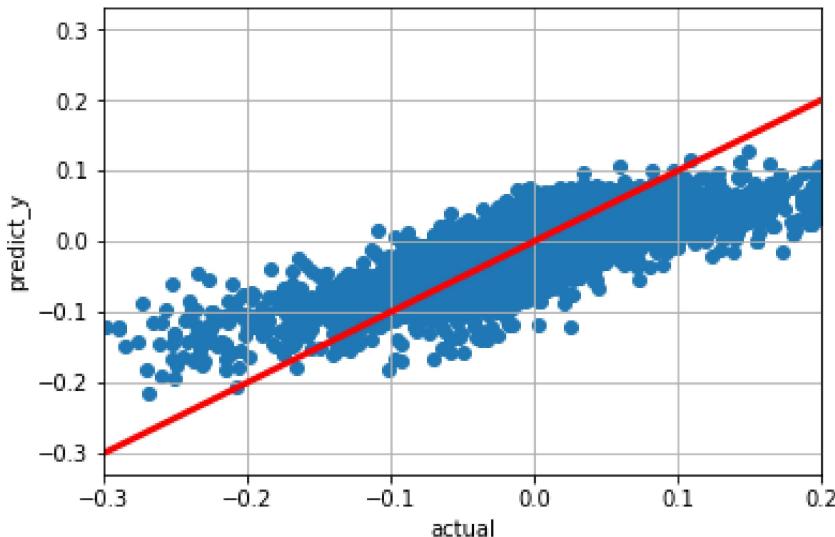


Create a scatter plot of the predicted vs. actual hand motion on the test data. On the same plot, plot the line where $yts_hat = yts$.

In [13]:

```
# TODO

#print(yts.shape,y_ts_pred.shape)
plt.scatter(yts,y_ts_pred)
plt.plot([-0.3,0.3], [-0.3,0.3], 'r-', linewidth=3)
plt.xlim([-0.3,0.2])
plt.xlabel('actual')
plt.ylabel('predict_y')
plt.grid()
plt.show()
```



Using K-fold cross validation for the optimal number of neurons

In the above, we fixed $d=100$. We can use cross validation to try to determine the best number of neurons to use. Try model orders with $d=10, 20, \dots, 190$. For each value of d , use K-fold validation with 10 folds to estimate the test RSS. For a data set this size, each fold will take a few seconds to compute, so it may be useful to print the progress.

In [14]:

```
import sklearn.model_selection

# Create a k-fold object
nfold = 10
kf = sklearn.model_selection.KFold(n_splits=nfold,shuffle=True)

# Model orders to be tested
dtest = np.arange(10,200,10)
nd = len(dtest)
RSSts = np.zeros((nd,nfold))
RSStm = np.zeros((nd,nfold))

for i,d in enumerate(dtest):
    Isel = np.argsort(Rsq)[::-1][:d]
    Inotsel = np.argsort(Rsq)[::-1][d:]
    #print(Isel.shape,Inotsel.shape)

    xdat = X0[:,Isel]
    ydat = y0[:]
    print(xdat.shape,ydat.shape)

    # Loop over the folds

    for isplit, Ind in enumerate(kf.split(xdat)):
        # train , test

        # Get the training data in the split
        Itr, Its = Ind
        xtr = xdat[Itr]

        ytr = ydat[Itr]
        xts = xdat[Its]
        yts = ydat[Its]

        # Fit data on training data
        regr.fit(xtr,ytr)
        print(xtr.shape)

        # Measure RSS on test data
        yhat = regr.predict(xts)

        RSSts[i,isplit] = np.mean((yhat-yts)**2)
        RSStm[i,isplit] = RSSts[i,isplit]/(np.std(yts)**2)
        #print(i,isplit,RSSts[i,isplit],RSStm[i,isplit])

    # TODO.
```

(15536, 10) (15536,)
(13982, 10)
(13982, 10)
(13982, 10)
(13982, 10)
(13982, 10)
(13982, 10)
(13983, 10)
(13983, 10)
(13983, 10)
(13983, 10)
(15536, 20) (15536,)
(13982, 20)
(13982, 20)
(13982, 20)
(13982, 20)
(13982, 20)
(13982, 20)
(13982, 20)
(13983, 20)
(13983, 20)
(13983, 20)
(13983, 20)
(13983, 20)
(13983, 20)
(15536, 30) (15536,)
(13982, 30)
(13982, 30)
(13982, 30)
(13982, 30)
(13982, 30)
(13982, 30)
(13982, 30)
(13983, 30)
(13983, 30)
(13983, 30)
(13983, 30)
(13983, 30)
(13983, 30)
(15536, 40) (15536,)
(13982, 40)
(13982, 40)
(13982, 40)
(13982, 40)
(13982, 40)
(13982, 40)
(13983, 40)
(13983, 40)
(13983, 40)
(13983, 40)
(13983, 40)
(15536, 50) (15536,)
(13982, 50)
(13982, 50)
(13982, 50)
(13982, 50)
(13982, 50)
(13982, 50)
(13983, 50)
(13983, 50)
(13983, 50)
(13983, 50)
(15536, 60) (15536,)
(13982, 60)

(13982, 60)
(13982, 60)
(13982, 60)
(13982, 60)
(13982, 60)
(13983, 60)
(13983, 60)
(13983, 60)
(13983, 60)
(15536, 70) (15536,)
(13982, 70)
(13982, 70)
(13982, 70)
(13982, 70)
(13982, 70)
(13982, 70)
(13982, 70)
(13982, 70)
(13983, 70)
(13983, 70)
(13983, 70)
(13983, 70)
(13983, 70)
(15536, 80) (15536,)
(13982, 80)
(13982, 80)
(13982, 80)
(13982, 80)
(13982, 80)
(13982, 80)
(13982, 80)
(13983, 80)
(13983, 80)
(13983, 80)
(13983, 80)
(13983, 80)
(13983, 80)
(15536, 90) (15536,)
(13982, 90)
(13982, 90)
(13982, 90)
(13982, 90)
(13982, 90)
(13982, 90)
(13982, 90)
(13983, 90)
(13983, 90)
(13983, 90)
(13983, 90)
(13983, 90)
(15536, 100) (15536,)
(13982, 100)
(13982, 100)
(13982, 100)
(13982, 100)
(13982, 100)
(13982, 100)
(13983, 100)
(13983, 100)
(13983, 100)
(13983, 100)
(15536, 110) (15536,)
(13982, 110)
(13982, 110)
(13982, 110)

(13982, 110)
(13982, 110)
(13982, 110)
(13983, 110)
(13983, 110)
(13983, 110)
(15536, 120) (15536,)
(13982, 120)
(13982, 120)
(13982, 120)
(13982, 120)
(13982, 120)
(13982, 120)
(13982, 120)
(13982, 120)
(13983, 120)
(13983, 120)
(13983, 120)
(13983, 120)
(13983, 120)
(13983, 120)
(15536, 130) (15536,)
(13982, 130)
(13982, 130)
(13982, 130)
(13982, 130)
(13982, 130)
(13982, 130)
(13982, 130)
(13983, 130)
(13983, 130)
(13983, 130)
(13983, 130)
(13983, 130)
(15536, 140) (15536,)
(13982, 140)
(13982, 140)
(13982, 140)
(13982, 140)
(13982, 140)
(13982, 140)
(13983, 140)
(13983, 140)
(13983, 140)
(13983, 140)
(13983, 140)
(15536, 150) (15536,)
(13982, 150)
(13982, 150)
(13982, 150)
(13982, 150)
(13982, 150)
(13982, 150)
(13983, 150)
(13983, 150)
(13983, 150)
(13983, 150)
(15536, 160) (15536,)
(13982, 160)
(13982, 160)
(13982, 160)
(13982, 160)
(13982, 160)

```
(13982, 160)
(13983, 160)
(13983, 160)
(13983, 160)
(13983, 160)
(15536, 170) (15536,)
(13982, 170)
(13982, 170)
(13982, 170)
(13982, 170)
(13982, 170)
(13982, 170)
(13983, 170)
(13983, 170)
(13983, 170)
(13983, 170)
(15536, 180) (15536,)
(13982, 180)
(13982, 180)
(13982, 180)
(13982, 180)
(13982, 180)
(13982, 180)
(13983, 180)
(13983, 180)
(13983, 180)
(13983, 180)
(13983, 180)
(13983, 180)
(15536, 190) (15536,)
(13982, 190)
(13982, 190)
(13982, 190)
(13982, 190)
(13982, 190)
(13982, 190)
(13983, 190)
(13983, 190)
(13983, 190)
(13983, 190)
```

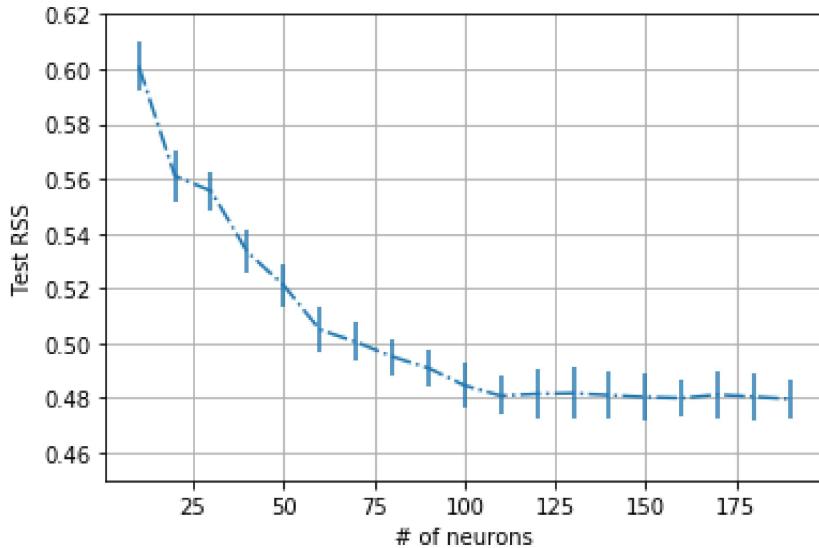
Compute the RSS test mean and standard error and plot them as a function of the model order d using the plt.errorbar() method.

In [15]:

```
# TODO
#print(RSStm)
RSS_mean = np.mean(RSStm, axis=1)
print(RSS_mean)
RSS_std = np.std(RSStm, axis=1) / np.sqrt(nfold-1)

plt.errorbar(dtest, RSS_mean, yerr=RSS_std, fmt='.-')
plt.xlabel('# of neurons')
plt.ylabel('Test RSS')
plt.ylim(0.45,0.62)
plt.grid()
```

```
[ 0.60115244  0.5610449   0.55549035  0.53330513  0.52114255  0.50495
 071
 0.50043811  0.49508058  0.49088173  0.4845564   0.48081938  0.48149
 978
 0.48176009  0.48100855  0.48033911  0.47995059  0.48122271  0.48042
 42
 0.47961102]
```



Find the optimal order using the one standard error rule. Print the optimal value of d and the mean test RSS per sample at the optimal d.

In [16]:

```
# TODO

imin = np.argmin(RSS_mean)
print("The selected model order is {0:d}".format(dtest[imin]))
RSStarget = np.mean(RSStm[imin,:])
print("The mean test RSS per sample is {0:f}".format(RSStarget))
```

```
The selected model order is 190
The mean test RSS per sample is 0.479611
```

Using LASSO regression

Instead of using the above heuristic to select the variables, we can use LASSO regression.

First use the `preprocessing.scale` method to standardize the data matrix X_0 . Store the standardized values in X_s . You do not need to standardize the response. For this data, the `scale` routine may throw a warning that you are converting data types. That is fine.

In [17]:

```
import sklearn.model_selection
from sklearn import datasets, linear_model, preprocessing

# TODO

Xs = preprocessing.scale(X0)
print(Xs.shape)

(15536, 196)

g:\code\python\lib\site-packages\sklearn\utils\validation.py:444: DataConversionWarning: Data with input dtype uint8 was converted to float64 by the scale function.
  warnings.warn(msg, DataConversionWarning)
```

Now, use the LASSO method to fit a model. Use cross validation to select the regularization level `alpha`. Use `alpha` values logarithmically spaced from `1e-5` to `0.1`, and use 10 fold cross validation.

In [18]:

```
# TODO

# Create a k-fold cross validation object
nfold = 10
kf = sklearn.model_selection.KFold(n_splits=nfold,shuffle=True)

# Create the LASSO model. We use the `warm start` parameter so that the fit will
# start at the previous value.
# This speeds up the fitting.
model = linear_model.Lasso(warm_start=True)

# Regularization values to test
nalpha = 20
alphas = np.logspace(-5,-1,nalpha)
print(alphas)

# MSE for each alpha and fold value
mse = np.zeros((nalpha,nfold))

for ifold, ind in enumerate(kf.split(Xs)):

    # Get the training data in the split
    Itr,Its = ind
    X_tr = Xs[Itr,:]
    y_tr = y0[Itr]
    X_ts = Xs[Its,:]
    y_ts = y0[Its]

    # Compute the Lasso path for the split
    for ia, a in enumerate(alphas):

        # Fit the model on the training data
        model.alpha = a
        model.fit(X_tr,y_tr)

        # Compute the prediction error on the test data
        y_ts_pred = model.predict(X_ts)
        mse[ia,ifold] = np.mean((y_ts_pred-y_ts)**2) / (np.std(yts)**2)

#print(mse)
```

```
[ 1.0000000e-05  1.62377674e-05  2.63665090e-05  4.28133240e-05
 6.95192796e-05  1.12883789e-04  1.83298071e-04  2.97635144e-04
 4.83293024e-04  7.84759970e-04  1.27427499e-03  2.06913808e-03
 3.35981829e-03  5.45559478e-03  8.85866790e-03  1.43844989e-02
 2.33572147e-02  3.79269019e-02  6.15848211e-02  1.0000000e-01]
```

Plot the mean test RSS and test RSS standard error with the plt.errorbar plot.

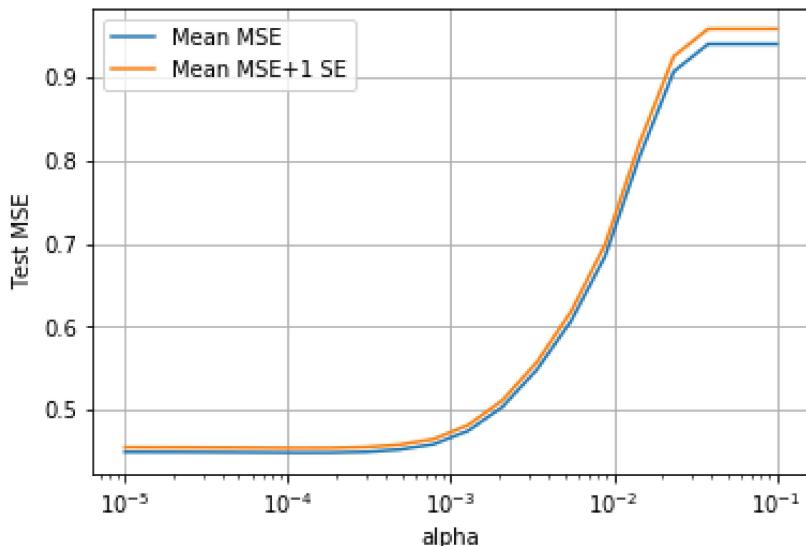
In [19]:

```
# TODO
print(mse.shape)

mse_mean = np.mean(mse, axis=1)
mse_std = np.std(mse, axis=1) / np.sqrt(nfold-1)

# Plot the mean MSE and the mean MSE + 1 std dev
plt.semilogx(alphas, mse_mean)
plt.semilogx(alphas, mse_mean+mse_std)
plt.legend(['Mean MSE', 'Mean MSE+1 SE'], loc='upper left')
plt.xlabel('alpha')
plt.ylabel('Test MSE')
plt.grid()
plt.show()
```

(20, 10)



Find the optimal alpha and mean test RSS using the one standard error rule.

In [20]:

```
# Find the minimum MSE and MSE target
imin = np.argmin(mse_mean)
mse_tgt = mse_mean[imin] + mse_std[imin]
alpha_min = alphas[imin]

# Find the least complex model with mse_mean < mse_tgt
I = np.where(mse_mean < mse_tgt)[0]
iopt = I[-1]
alpha_opt = alphas[iopt]
print("Optimal alpha = %f" % alpha_opt)

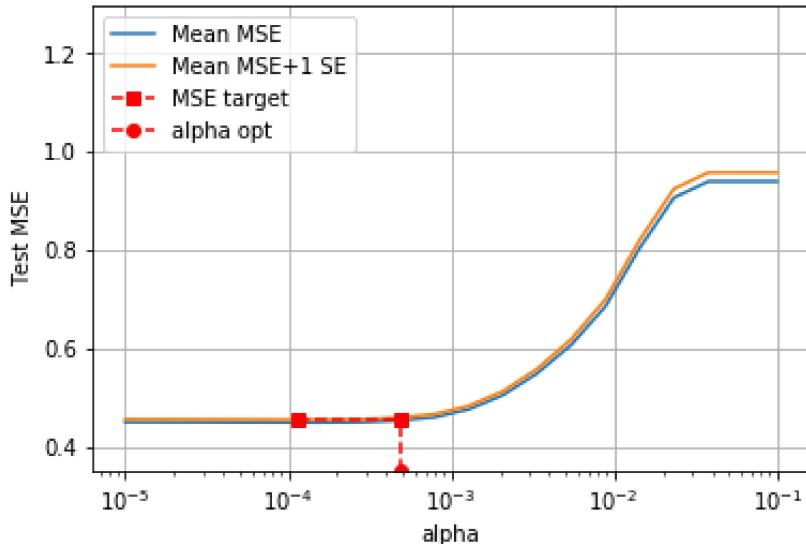
# Plot the mean MSE and the mean MSE + 1 std dev
plt.semilogx(alphas, mse_mean)
plt.semilogx(alphas, mse_mean+mse_std)

# Plot the MSE target
plt.semilogx([alpha_min,alpha_opt], [mse_tgt,mse_tgt], 'rs--')

# Plot the optimal alpha line
plt.semilogx([alpha_opt,alpha_opt], [0.35,mse_mean[iopt]], 'ro--')

plt.legend(['Mean MSE', 'Mean MSE+1 SE', 'MSE target','alpha opt'],loc='upper left')
plt.xlabel('alpha')
plt.ylabel('Test MSE')
plt.ylim([0.35,1.3])
plt.grid()
plt.show()
```

Optimal alpha = 0.000483



Using the optimal alpha, recompute the predicted response variable on the whole data. Plot the predicted vs. actual values.

In [21]:

```
# TODO

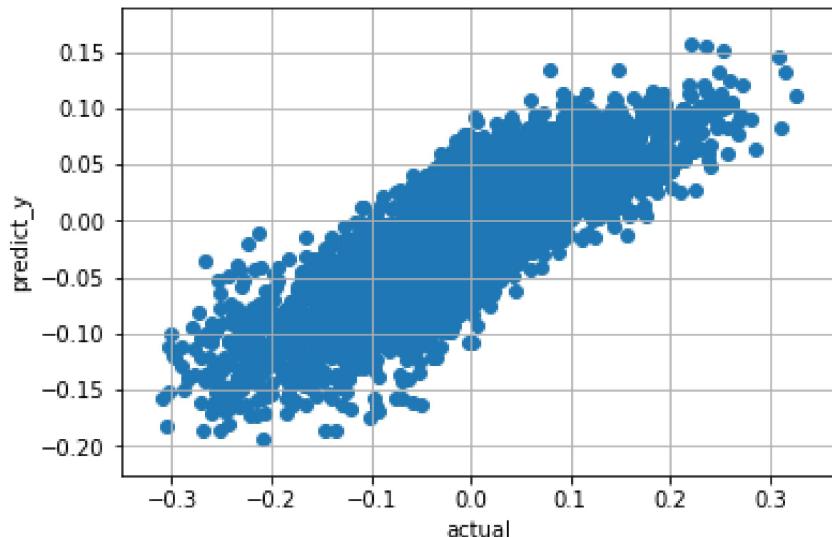
model.alpha = alpha_opt
model.fit(Xs,y0)

# Print the coefficients
print(model.coef_.shape)

y_ts_pred = model.predict(Xs)

plt.scatter(y0,y_ts_pred)
# plt.plot([-0.3,0.3], [-0.3,0.3], 'r-', Linewidth=3)
# plt.xlim([-0.3,0.2])
plt.xlabel('actual')
plt.ylabel('predict_y')
plt.grid()
plt.show()
```

(196,)



More Fun

You can play around with this and many other neural data sets. Two things that one can do to further improve the quality of fit are:

- Use more time lags in the data. Instead of predicting the hand motion from the spikes in the previous time, use the spikes in the last few delays.
- Add a nonlinearity. You should see that the predicted hand motion differs from the actual for high values of the actual. You can improve the fit by adding a nonlinearity on the output. A polynomial fit would work well here.

You do not need to do these, but you can try them if you like.