Elaina-4-neural-model-selection-hw

July 5, 2021

1 Model Order Selection for Neural Data

***italicized text*Name**: [Elaina(Yaogui) Huang]
Net ID: [yh4310]

2 Part I

Part I contains my answer for the PDF problem set.

a.

wt = = <1, 2, 0>

b.

Not possible to express the true function

c.

wt = = <0, 1, 2, 3>

d.

Not possible to express the true function

e.

wt = = <1, -3, 3, -1>

3 Part II

Part II contains original colab notebook.

Attribution: This notebook is a slightly adapted version of the model order selection lab assignment 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.

3.1 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

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.

[ ]:
```
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. However, the raw data files can be quite large. To make the lab easier, the Kording lab at UPenn has put together an excellent repository where they have created simple pre-processed versions of the data. You can download the file `example_data_s1.pickle` from the Dropbox link. Alternatively, you can directly run the following command. This may take a little while to download since the file is 26 MB.

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

```
--2021-07-02 19:00:16--  https://www.dropbox.com/sh/n4924ipcfjqc0t6/AADOv9JYMUBK
1tlg9P71gSSra/example_data_s1.pickle?dl=1
Resolving www.dropbox.com (www.dropbox.com)... 162.125.5.18,
2620:100:601d:18::a27d:512
Connecting to www.dropbox.com (www.dropbox.com)|162.125.5.18|:443... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location:
/sh/dl/n4924ipcfjqc0t6/AADOv9JYMUBK1tlg9P71gSSra/example_data_s1.pickle
[following]
--2021-07-02 19:00:16--  https://www.dropbox.com/sh/dl/n4924ipcfjqc0t6/AADOv9JYM
UBK1tlg9P71gSSra/example_data_s1.pickle
Reusing existing connection to www.dropbox.com:443.
HTTP request sent, awaiting response... 302 Found
Location:
```

```
https://ucc3f9a894c2656ec8db139edaed.dl.dropboxusercontent.com/cd/0/get
/BRjYksHW7pVdcRsEg5wztE4gjlyTmBqrnoUTI8LVMjMDObAsY64X7ASljWX-
o4m1QjUMp6DSk0bpy7HSrBsdeM5qA6s21jjHYxPoYLQRfPccRCzmLGuiNT0PnAkzLKR18SjMniT-
MoxbwRi7CXOCkls8/file?dl=1# [following]
--2021-07-02 19:00:17--
https://ucc3f9a894c2656ec8db139edaed.dl.dropboxusercontent.com/cd/0/get
/BRjYksHW7pVdcRsEg5wztE4gjlyTmBqrnoUTI8LVMjMDObAsY64X7ASljWX-
o4m1QjUMp6DSk0bpy7HSrBsdeM5qA6s21jjHYxPoYLQRfPccRCzmLGuiNT0PnAkzLKR18SjMniT-
MoxbwRi7CXOCkls8/file?dl=1
Resolving ucc3f9a894c2656ec8db139edaed.dl.dropboxusercontent.com
(ucc3f9a894c2656ec8db139edaed.dl.dropboxusercontent.com)... 162.125.5.15,
2620:100:601d:15::a27d:50f
Connecting to ucc3f9a894c2656ec8db139edaed.dl.dropboxusercontent.com
(ucc3f9a894c2656ec8db139edaed.dl.dropboxusercontent.com)|162.125.5.15|: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  51.7MB/s    in 0.5s

2021-07-02 19:00:18 (51.7 MB/s) - example_data_s1.pickle saved
[26498656/26498656]
```

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.

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

[ ]:
```
X
```

[ ]:
```
array([[0., 0., 0., ..., 0., 0., 1.],
       [0., 0., 1., ..., 0., 0., 1.],
       [0., 0., 0., ..., 0., 0., 1.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 1., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])
```

[ ]:
```
y
```

[ ]:
```
array([[ 1.51037413e-01,  1.50912405e-01],
       [-1.39498351e-01,  1.10064258e-01],
       [-3.55773944e-01, -3.96432704e-01],
       ...,
       [ 3.19722904e-06, -1.32722761e-06],
       [ 7.24814520e-07,  3.01404519e-06],
```

```
       [ 9.51148351e-07, -1.48976504e-06]])
```

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 in time bin `i` * `y[i,1]` = velocity of the monkey's hand in the y-direction in time bin `i`

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.

[ ]:
```
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)
```

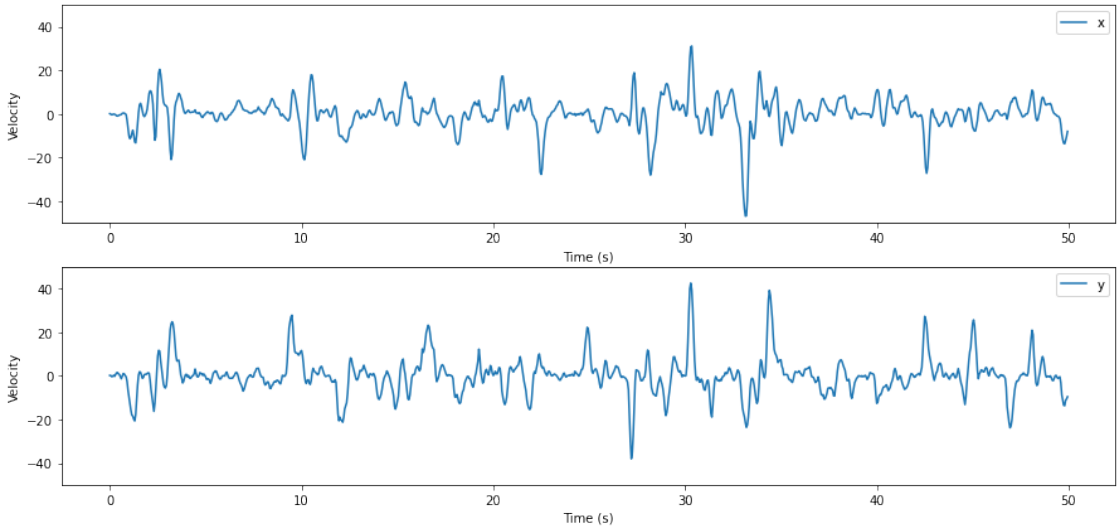
```
Number of neurons = 52
Number of time samples = 61339
Number of outputs = 2
Total time (secs) = 3066.950000
```

Then, we can plot the velocity against time, for each direction, for the first 1000 samples:

[ ]:
```
t_cutoff = 1000
directions = ['x', 'y']

fig, axes = plt.subplots(nrows=2, ncols=1, figsize=(15,7))
for n in range(nout):
  sns.lineplot(x=np.arange(0, t_cutoff)*tsamp, y=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)
```



We can also "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.

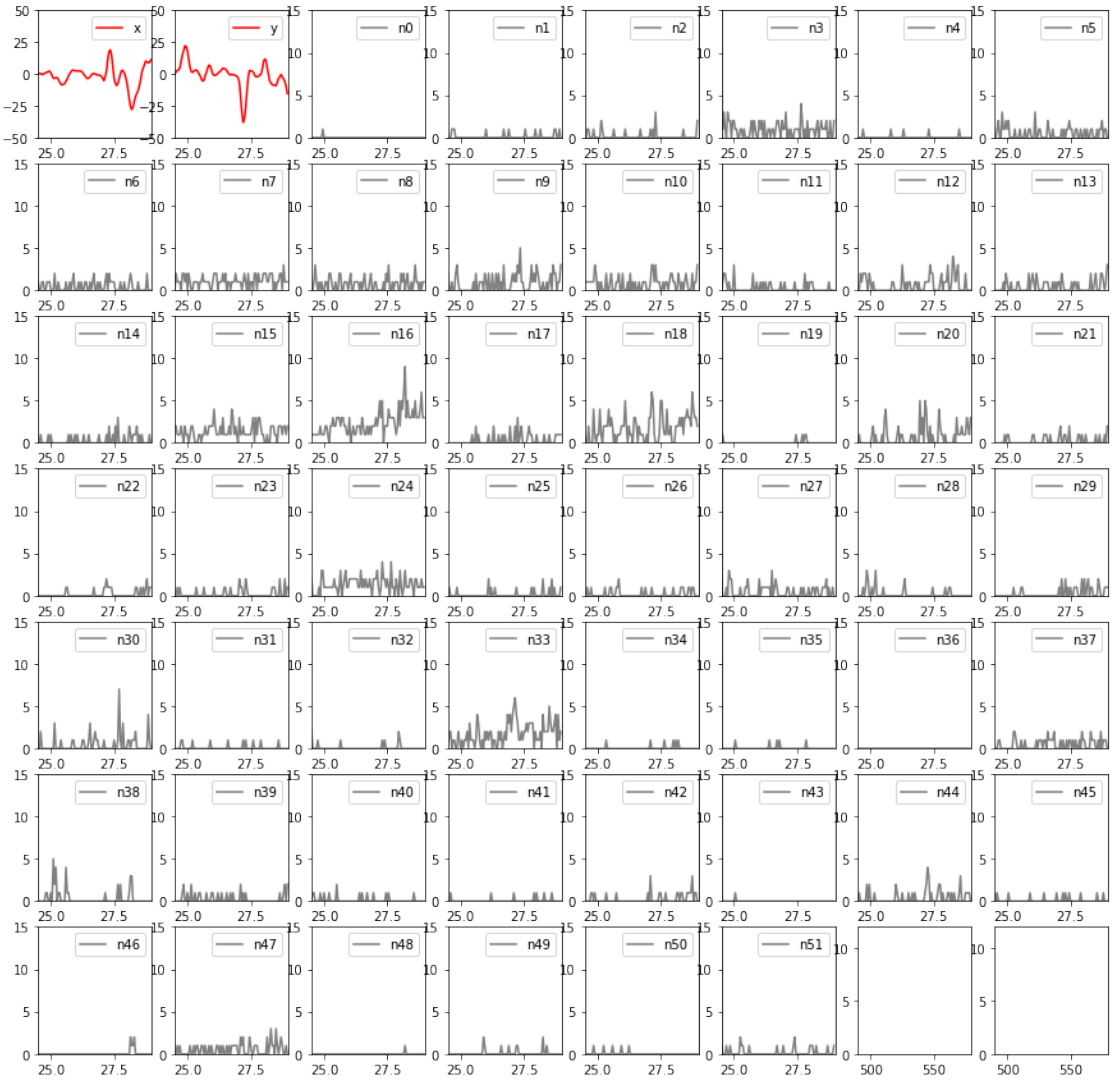
[ ]:
```
t_start = 490
t_end = 580

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(x=np.arange(t_start, t_end)*tsamp, y=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(x=np.arange(t_start, t_end)*tsamp, y=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));
```



3.2 Fitting a linear model

Let's first try a 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, 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.

[ ]:
```
# TODO 1
# Xtr, Xts, ytr, yts = ...
Xtr, Xts = train_test_split(X, test_size=0.33, shuffle=False)
ytr, yts = train_test_split(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 sklearn `r2_score` method. Print the `rsq` value. You should get `rsq` of around `0.45`.

[ ]:
```
# TODO 2
reg = LinearRegression().fit(Xtr, ytr)
yhat = reg.predict(Xts)
rsq = r2_score(yts, yhat)
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,

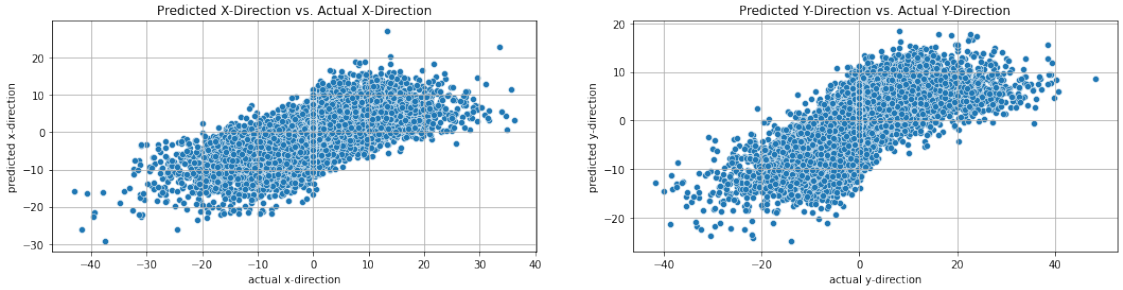
- one of predicted X direction vs. actual X direction,
- one of predicted Y direction vs. actual Y direction

Make sure to carefully label each axis.

[ ]:
```
# TODO 3A
fig = plt.figure(figsize=(18,4))

plt.subplot(1,2,1)
sns.scatterplot(data=ytr, x=yts[:,0], y=yhat[:,0]);
plt.title('Predicted X-Direction vs. Actual X-Direction')
plt.xlabel('actual x-direction')
plt.ylabel('predicted x-direction')
plt.grid()

plt.subplot(1,2,2)
sns.scatterplot(data=ytr, x=yts[:, 1], y=yhat[:, 1]);
plt.title('Predicted Y-Direction vs. Actual Y-Direction')
plt.xlabel('actual y-direction')
plt.ylabel('predicted y-direction')
plt.grid()
```



It can also be useful to visualize the actual and predicted values over time, for a slice of time. Create two subplots, both with time on the horizontal axis, but only including *the first 1000 rows* in the data. On the vertical axis,

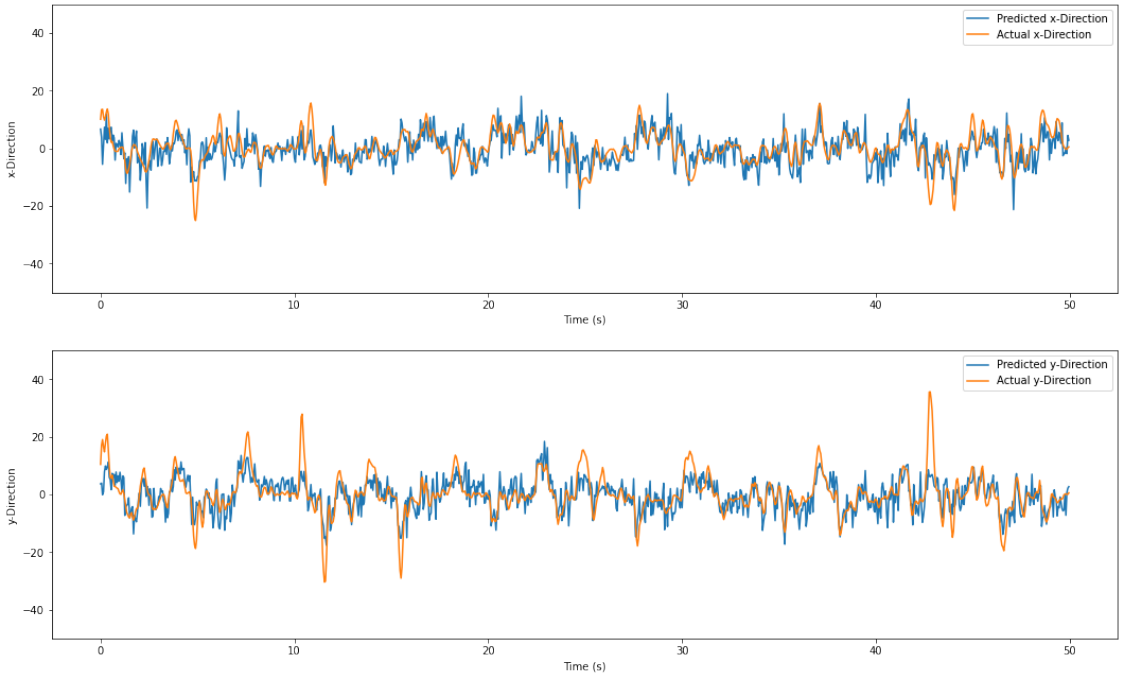
- for one subplot: show the actual X direction as a line of one color, and the predicted X direction as a line of another color.
- for the second subplot: show the actual Y direction as a line of one color, and the predicted Y direction as a line of another color.

Make sure to carefully label each axis (including units on the time axis!), and label the data series (i.e. which color is the actual value and which is the predicted value).

[ ]:
```
# TODO 3B

fig, axes = plt.subplots(nrows=2, ncols=1, figsize=(18,11))
for n in range(nout):
  sns.lineplot(x=np.arange(0, t_cutoff)*tsamp, y=yhat[0:t_cutoff, n],␣
 ↪label="Predicted "+directions[n]+"-Direction", ax=axes[n]);
  sns.lineplot(x=np.arange(0, t_cutoff)*tsamp, y=yts[0:t_cutoff, n],␣
 ↪label="Actual "+directions[n]+"-Direction", ax=axes[n]);
```

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



Comment on this plot - does the model predict the hand velocity well?

comments: overall, it seems like the prediction is doing an OK job catching the patterns, however, there are several high peak or low botton points in the actual data pattern that the prediction did not catch them at all. There are other times when the actual data did not have that pattern but the prediction show the pattern. In fact, the R2 score has shown the explained data is only around 45%, that's below half.

3.3 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 ith 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 $i-$ dly (where 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.

[ ]:
```
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:

[ ]:
```
X_dly1, y_dly1 = create_dly_data(X, y, 1)
```

Verify that the dimensions have changed, as expected:

[ ]:
```
# dimensions of original data matrix
X.shape
```

[ ]:
```
(61339, 52)
```

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

[ ]:
```
(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.)

[ ]:
```
X_dly1[0]
```

[ ]:
```
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.])
```

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

[ ]:
```
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.])
```

[ ]:
```
y_dly1[0]
```

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

[ ]:
```
y[1]
```

[ ]:
```
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 (again, do not shuffle the data) * Fit the model on the training data * Measure the R2 score on the test data

If you did this correctly, you should get a new R2 score around 0.60. This is significantly better than the memoryless model.

[ ]:
```
# TODO 4

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

# Split into training and test
Xdly_tr, Xdly_ts, ydly_tr, ydly_ts = train_test_split(Xdly, ydly, test_size=0.
 ↪33, shuffle=False)

# Create linear regression object
# Fit the model
reg_dly = LinearRegression().fit(Xdly_tr, ydly_tr)
yhat_dly = reg_dly.predict(Xdly_ts)

# Measure the new r2 score
rsq_dly = r2_score(ydly_ts, yhat_dly)
rsq_dly
```

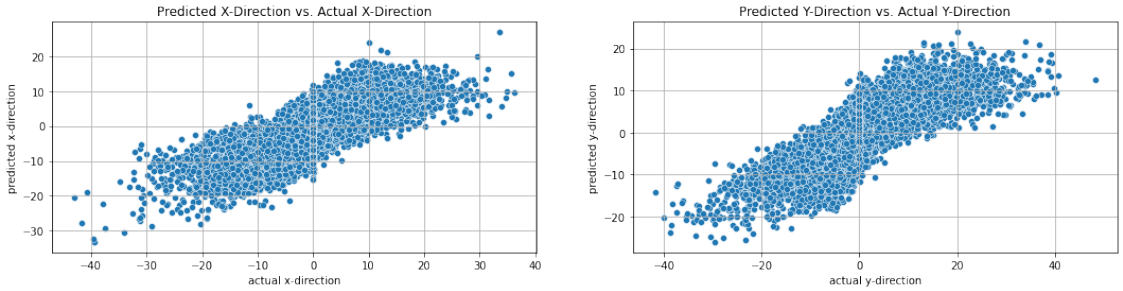
[ ]:
```
0.6033897697058304
```

Plot the predicted vs. true values as before, with one subplot for X velocity and one plot for Y velocity.

[ ]:
```
# TODO 5A
fig = plt.figure(figsize=(18,4))

plt.subplot(1,2,1)
sns.scatterplot(data=ydly_tr, x=ydly_ts[:,0], y=yhat_dly[:,0]);
plt.title('Predicted X-Direction vs. Actual X-Direction')
plt.xlabel('actual x-direction')
plt.ylabel('predicted x-direction')
plt.grid()

plt.subplot(1,2,2)
sns.scatterplot(data=ydly_tr, x=ydly_ts[:,1], y=yhat_dly[:,1]);
plt.title('Predicted Y-Direction vs. Actual Y-Direction')
plt.xlabel('actual y-direction')
plt.ylabel('predicted y-direction')
plt.grid()
```



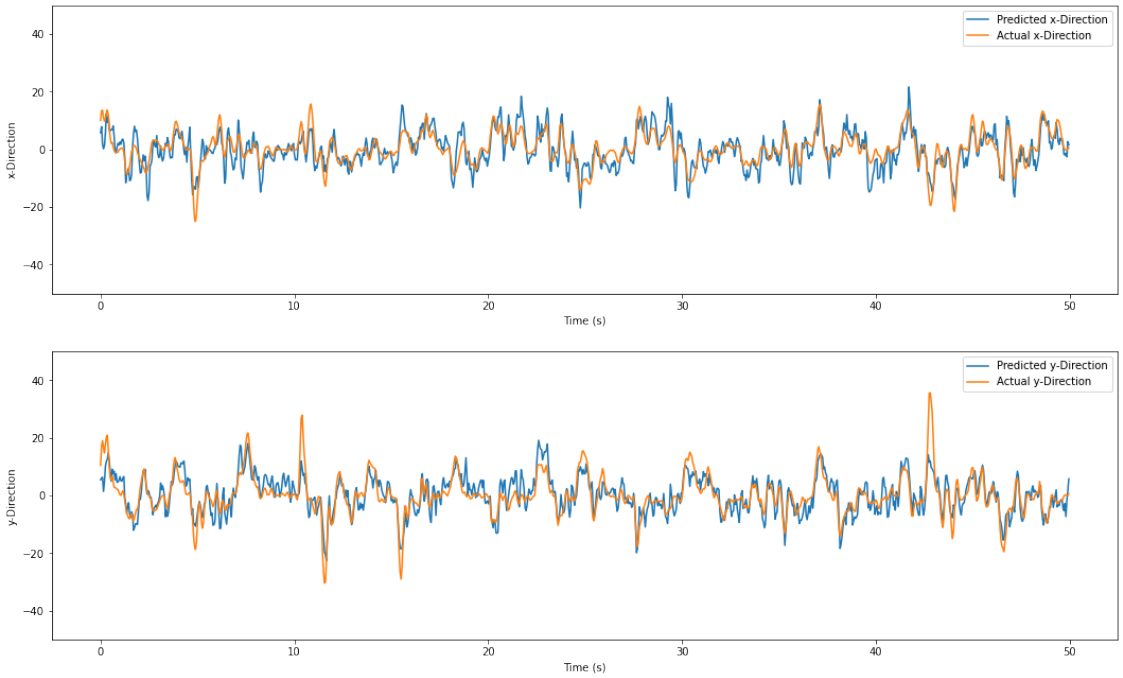
Also plot the actual and predicted values over time for the first 1000 samples, as you did before. Comment on this plot - does the model predict the hand velocity well?

comment: Compare to the last model, this prediction seems to be able to cover the actual data pattern more closely. Previously, several high peak/low botton points were not covered by the predction model, it is still not fully covered in here but it has shown a great improvement.

[ ]:
```
# TODO 5B

fig, axes = plt.subplots(nrows=2, ncols=1, figsize=(18,11))
for n in range(nout):
  sns.lineplot(x=np.arange(0, t_cutoff)*tsamp, y=yhat_dly[0:t_cutoff, n],␣
 ↪label="Predicted "+directions[n]+"-Direction", ax=axes[n]);
  sns.lineplot(x=np.arange(0, t_cutoff)*tsamp, y=ydly_ts[0:t_cutoff, n],␣
 ↪label="Actual "+directions[n]+"-Direction", ax=axes[n]);

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



3.4 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.

[ ]:
```
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`.

[ ]:
```
dmax = 15

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

[ ]:
```
Xdly.shape
```

[ ]:
```
(5985, 832)
```

[ ]:
```
ydly.shape
```

[ ]:
```
(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:

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

[ ]:
```
(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`:

[ ]:
```
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]
```

[ ]:
```
Xdly.shape
```

[ ]:
```
(5985, 832)
```

[ ]:
```
X_dtest.shape
```

[ ]:
```
(5985, 104)
```

[ ]:
```
Xdly
```

[ ]:
```
array([[0., 1., 0., ..., 0., 0., 1.],
       [0., 0., 0., ..., 0., 0., 1.],
       [0., 0., 1., ..., 0., 0., 1.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 1., ..., 0., 0., 0.],
       [0., 0., 1., ..., 0., 0., 0.]])
```

You can refer to the example in the "Model order selection" section of the demo notebook. But, make sure to use `shuffle=False` in your `KFold` object, since for this example it would be inappropriate to shuffle the data.

[ ]:
```
# Number of folds
nfold = 10

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

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

# Loop over the folds
for i, idx_split in enumerate(kf.split(Xdly)):
```

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

    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[:,:(dtest+1)*nneuron] #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 = X_dtest[idx_tr]
        ytr = ydly[idx_tr]
        Xval = X_dtest[idx_val]
        yval = ydly[idx_val]

        # TODO 10  Fit linear regression on training data
        reg_dly_sampled = LinearRegression().fit(Xtr, ytr)
        yhat_dly_sampled = reg_dly_sampled.predict(Xval)

        # TODO 11  Measure the R2 on validation data and store in the matrix␣
↪Rsq
        rsq_dly_samples = r2_score(yval, yhat_dly_sampled)
        Rsq[it, i] = rsq_dly_samples
print(Rsq)
```

```
[[0.45197502 0.43227267 0.4846026  0.43112577 0.40030795 0.38615897
  0.44228733 0.43414185 0.47665037 0.46100337]
 [0.55473725 0.55677757 0.57600486 0.55382912 0.51132554 0.51363154
  0.54634043 0.54915035 0.58459043 0.57731981]
 [0.59229587 0.61287861 0.61194363 0.61233195 0.55782397 0.56546394
  0.59210925 0.59136745 0.61920178 0.61654828]
 [0.61403143 0.6357882  0.63506292 0.65096342 0.58072703 0.58360049
  0.61905936 0.61035842 0.6378388  0.63606118]
 [0.6259339  0.64754302 0.65531661 0.67511442 0.59326144 0.60242113
  0.64044374 0.621237   0.65706291 0.65073685]
 [0.64371528 0.65387454 0.66843527 0.68784184 0.59932914 0.62200826
  0.65911435 0.62633418 0.67638756 0.66139106]
 [0.65493271 0.65793096 0.67310461 0.69542598 0.60704193 0.63779232
  0.67203623 0.62793397 0.68747742 0.6702819 ]
 [0.65744004 0.65990289 0.6732896  0.69651699 0.61092006 0.6477482
  0.68156933 0.63129558 0.69334639 0.67642195]
 [0.65908843 0.66397051 0.6736172  0.69568661 0.61456004 0.65190718
  0.6874251  0.63579036 0.69090095 0.68065806]
```

```
[0.66010189 0.66427715 0.6747861  0.69409131 0.61399284 0.65262564
 0.69099964 0.63744705 0.6826877  0.68162741]
[0.66229568 0.66311969 0.67677204 0.69322389 0.61016545 0.65524661
 0.69491201 0.63748292 0.6773267  0.67911056]
[0.66318676 0.66250513 0.67639795 0.69140643 0.60482579 0.66053674
 0.69574471 0.63503519 0.67439426 0.67428389]
[0.66117383 0.66189367 0.6744311  0.68699306 0.59847552 0.66059213
 0.69638465 0.63174299 0.67016592 0.67084991]
[0.65735187 0.66186261 0.67460722 0.68315794 0.59539878 0.66166963
 0.69570885 0.62812413 0.66614182 0.66755849]
[0.65450152 0.66127192 0.67363346 0.67970957 0.59125287 0.66414334
 0.69267588 0.62374952 0.66399571 0.66568454]
[0.65271703 0.6593568  0.6712717  0.67839723 0.58800809 0.6606374
 0.68891425 0.62135739 0.66155445 0.66383307]]
```

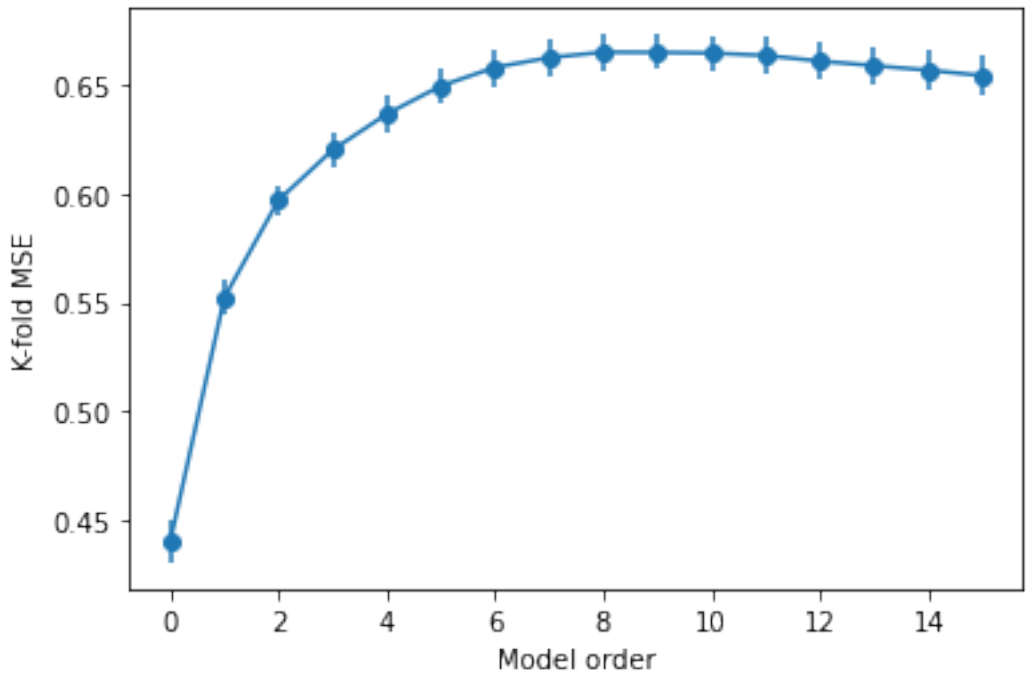
[ ]:
```
Rsq.shape
```

[ ]:
```
(16, 10)
```

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.

[ ]:
```
# TODO 12
Rsq_mean = Rsq.mean(axis=1)
Rsq_std  = Rsq.std(axis=1)/np.sqrt(nfold-1)
plt.errorbar(x=dtest_list, y=Rsq_mean, yerr=Rsq_std, marker='o');
plt.xlabel('Model order')
plt.ylabel('K-fold MSE')
```

[ ]:
```
Text(0, 0.5, 'K-fold MSE')
```



[ ]:
```
idx_max = np.argmax(Rsq_mean)
idx_max
```

[ ]:
```
8
```

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

[ ]:
```
# TODO 13
idx_max = np.argmax(Rsq_mean)
target_r2 = Rsq[idx_max,:].mean() - Rsq[idx_max,:].std()/np.sqrt(nfold-1)
idx_one_se_r2 = np.where(Rsq_mean > target_r2)
d_one_se_r2 = np.min(dtest_list[idx_one_se_r2])
d_one_se_r2
```

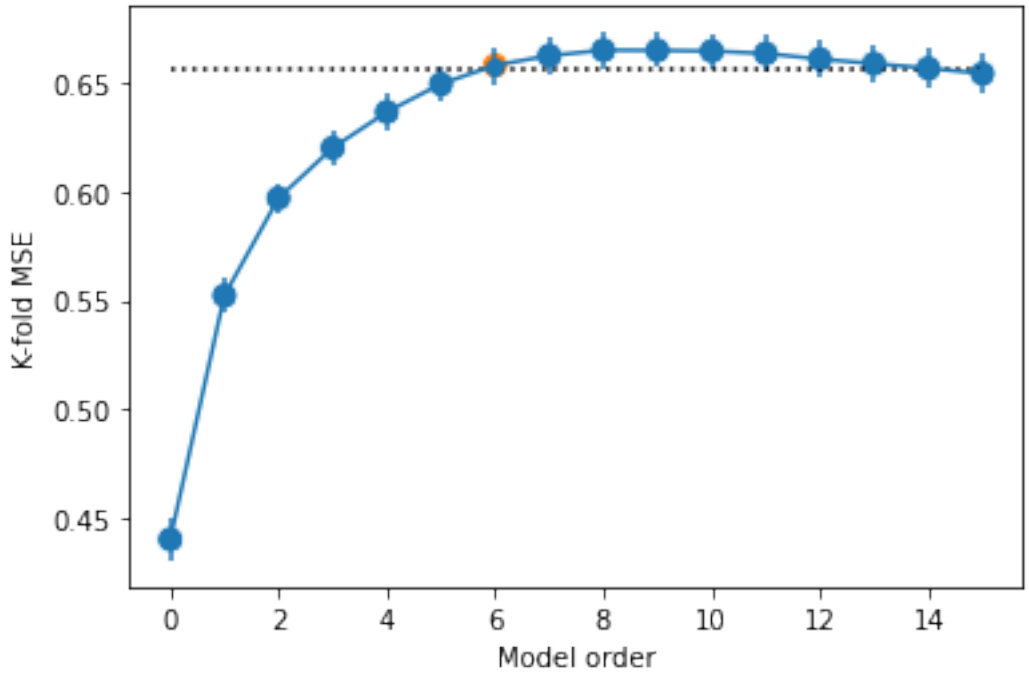
[ ]:
```
6
```

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 best R2). Print the best delay according to the "one SE rule."

[ ]:
```
# TODO 14
Rsq_mean = Rsq.mean(axis=1)
Rsq_std  = Rsq.std(axis=1)/np.sqrt(nfold-1)
plt.errorbar(x=dtest_list, y=Rsq_mean, yerr=Rsq_std, marker='o');
plt.hlines(y=target_r2, xmin=np.min(dtest_list), xmax=np.max(dtest_list),␣
 ↪ls='dotted')
sns.scatterplot(x=dtest_list, y=Rsq_mean, hue=dtest_list==d_one_se_r2, s=100,␣
 ↪legend=False);
```

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

[ ]:
```
Text(0, 0.5, 'K-fold MSE')
```



3.5 Fitting the selected model

[ ]:
```
Xdly.shape
```

[ ]:
```
(5985, 832)
```

Now that we have selected a model order, we can fit the (reduced) data to that model.

Use your `Xdly` and `ydly` to fit a linear regression model using the best delay according to the one SE rule.

[ ]:
```
# TODO 15
dtest_new = 6
X_dtest_new = Xdly[:,:(dtest_new+1)*nneuron]
y_dtest_new = ydly[:,:(dtest_new+1)*nneuron]

#Xdlytr, Xdlyts, ydlytr, ydlyts = train_test_split(X_dtest_new, y_dtest_new,␣
↪test_size=0.33, shuffle=False)
reg_dly_new = LinearRegression().fit(X_dtest_new, y_dtest_new)
```

[ ]:
```
Xts_dly2.shape
```

[ ]:
```
(1000, 364)
```

[ ]:
```
X_dtest_new.shape
```

[ ]:
```
(5985, 364)
```

Then, define a test set using data that was not used to train the model:

[ ]:
```
# TODO 16
# if dopt_one_se is the optimal model order, you can use
dopt_one_se = 6
Xts = X[nred+1:nred+1001+dopt_one_se]
yts = y[nred+1:nred+1001+dopt_one_se]
# and then use
Xts_dly2, yts_dly2 = create_dly_data(Xts,yts,dopt_one_se)
```

Use your fitted model to find the R2 score on the test set.

[ ]:
```
# TODO 17
yhatdly_new = reg_dly_new.predict(Xts_dly2)
rsq_dly2_ = r2_score(yts_dly2, yhatdly_new)
rsq_dly2_
```

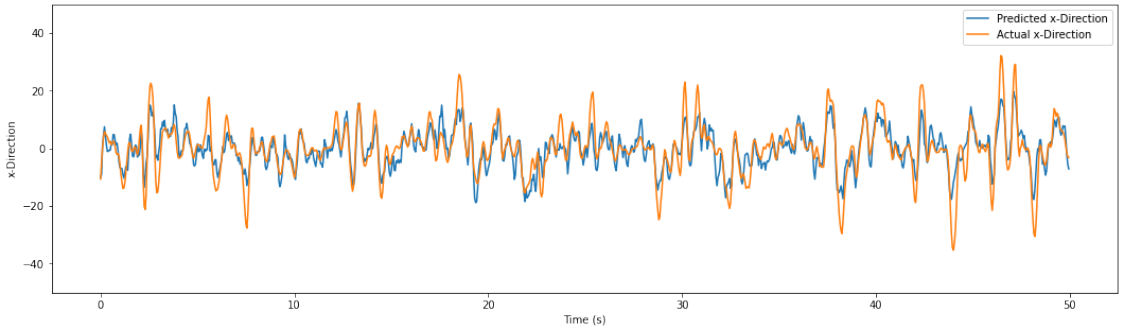
[ ]:
```
0.6949084442568592
```

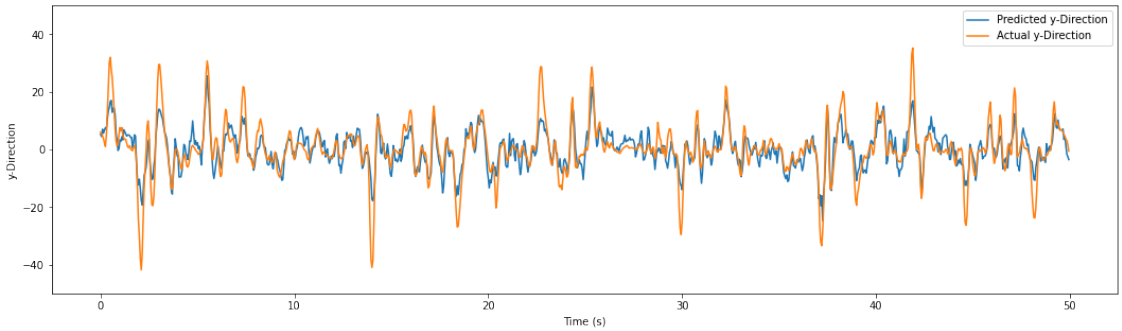
Also plot the actual and predicted values over time for the first 1000 samples of the *test* data (similar to your plots in the previous sections). Comment on this plot - does the model predict the hand velocity well?

comment: The R2 score has a great improvement, from 0.60 to 0.69. The new model, however, still cannot do very well when predicting extreme values, but in general the prediction better fitted the acutal line. Compare to the previous model, which missed many regular-range values(I mean the values that weren't extremely high or low), the current model is more reliable. Oddly, the previous model have a better match with the extreme values also because the extreme values in the previous value have a lower range than in the current model, may be due to the overlapping data created from the delay method that resulted in higher range thus potentially explain the current model's bas prediction at the pattern of extreme values.

[ ]:
```
# TODO 18
fig, axes = plt.subplots(nrows=2, ncols=1, figsize=(18,11))
for n in range(nout):
  sns.lineplot(x=np.arange(0, t_cutoff)*tsamp, y=yhatdly_new[0:t_cutoff, n],␣
 ↪label="Predicted "+directions[n]+"-Direction", ax=axes[n]);
  sns.lineplot(x=np.arange(0, t_cutoff)*tsamp, y=yts_dly2[0:t_cutoff, n],␣
 ↪label="Actual "+directions[n]+"-Direction", ax=axes[n]);

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


Predicted x-Direction
Actual x-Direction
x-Direction
Time (s)


Predicted y-Direction
Actual y-Direction
y-Direction
Time (s)