脑机智能导论HW2

HW2和附加作业代码均已放在https://github.com/ruanyijie/naoji中

任务1 神经元锋电位信号分析

1.数据处理

此部分主要参考了GitHub中DynamicalComponentsAnalysis的tutorial中的相关代码，从`chan_names`数据集中获取通道名称，然后通过解析获取 M1 和 S1 的索引，并获取采样时间相关数据。从`t`数据集中获取时间信息。 从`target_pos`数据集中获取目标位置信息。然后，通过计算目标位置的变化，确定每个试验的起始时间和结束时间。从`cursor_pos`数据集中经过处理获取速度。 将获取的时间、试验开始时间、试验结束时间、目标位置信息、达到距离（reach distance）和达到角度（reach angle）等信息整理成一个字典。

```
with h5py.File(filename, "r") as f:
        # Get channel names (e.g. M1 001 or S1 001)
        n_channels = f['chan_names'].shape[1]
        chan_names = []
        for i in range(n_channels):
            chan_names.append(f[f['chan_names'][0, i]][()].tobytes()
[::2].decode())
        # Get M1 and S1 indices
        M1_indices = [i for i in range(n_channels) if chan_names[i].split(' ')
[0] == 'M1']
        S1_indices = [i for i in range(n_channels) if chan_names[i].split(' ')
[0] == 'S1']
        # Get time
        t = f['t'][0, :]
        # Individually process M1 and S1 indices
        result = {}
        for indices in (M1_indices, S1_indices):
            if len(indices) == 0:
                continue
            # Get region (M1 or S1)
            region = chan_names[indices[0]].split(" ")[0]
            print(region)
            # Perform binning
            n_channels = len(indices)
            n_sorted_units = f["spikes"].shape[0] - 1  # The FIRST one is the
'hash' -- ignore!
            d = n_channels * n_sorted_units
            max_t = t[-1]
            min_t = t[0]
            binned_spikes = np.zeros((len(t), d), dtype=np.float32)
            for chan_idx in indices:
                for unit_idx in range(1, n_sorted_units + 1):  # ignore hash!
                    spike_times = f[f["spikes"][unit_idx, chan_idx]][()]
                    if spike_times.shape == (2,):
                        # ignore this case (no data)
                        continue
```

```
                    spike_times = spike_times[0, :]
                    # get rid of extraneous t vals
                    spike_times = spike_times[spike_times <= max_t]
                    spike_times = spike_times[spike_times >= min_t]
                    # make sure to ignore the hash here...
                    binned_spikes[:len(spike_times), chan_idx * n_sorted_units +
unit_idx - 1] = spike_times
            binned_spikes = binned_spikes[:,
np.count_nonzero(binned_spikes,axis=0) > thresh]

            result[region] = binned_spikes

        # Find when target pos changes
        target_pos = f["target_pos"][:].T
        import pandas as pd
        target_pos = pd.DataFrame(target_pos)
        has_change = target_pos.fillna(-1000).diff(axis=0).any(axis=1) # filling
NaNs with arbitrary scalar to treat as one block
        time = pd.DataFrame(f['t'][0, :].T)
        # Add start and end times to trial info
        change_times = time.index[has_change]
        start_times = change_times[:-1]
        end_times = change_times[1:]
        # Get target position per trial
        temp_target_pos = target_pos.loc[start_times].to_numpy().tolist()
        # Compute reach distance and angle
        reach_dist = target_pos.loc[end_times - 1].to_numpy() -
target_pos.loc[start_times - 1].to_numpy()
        reach_angle = np.arctan2(reach_dist[:, 1], reach_dist[:, 0]) / np.pi *
180

        # Create trial info
        result['time'] = time.to_numpy()
        # print(result['time'])
        result['start_times'] = start_times.to_numpy()
        result['end_times'] = end_times.to_numpy()
        result['target_pos'] = temp_target_pos
        result['reach_dist_x'] = reach_dist[:, 0]
        result['reach_dist_y'] = reach_dist[:, 1]
        result['reach_angle'] = reach_angle
        spike_time = []
        for indices in (M1_indices, S1_indices):
            for chan_idx in indices:
                    for unit_idx in range(1, n_sorted_units + 1):  # ignore
hash!
                        spike_times = f[f["spikes"][unit_idx, chan_idx]][()]
                        # if spike_times.shape == (2,):
                        #     # ignore this case (no data)
                        #     continue
                        # spike_times = spike_times[0, :]
                        spike_time.append(spike_times)

        result['spike_times'] = np.array(spike_time,dtype=object)
        print(result['spike_times'])
        for i in range (0,len(result['reach_angle'])):
            vel = []
            if(i==0):
```

```
                vel.append((result['reach_dist_x'][1]-result['reach_dist_x']
[0])/0.004)
                vel.append((result['reach_dist_y'][1]-result['reach_dist_y']
[0])/0.004)
            else:
                vel.append((result['reach_dist_x'][i]-result['reach_dist_x'][i-
1])/0.004)
                vel.append((result['reach_dist_x'][i]-result['reach_dist_x'][i-
1])/0.004)
            result['vels'].append(vel)
            if(i==0):
                result['vels'].append(math.sqrt((result['reach_dist_x'][1]-
result['reach_dist_x'][0])*(result['reach_dist_x'][1]-result['reach_dist_x']
[0])+(result['reach_dist_y'][1]-result['reach_dist_y'][0])*
(result['reach_dist_y'][1]-result['reach_dist_y'][0])/0.004))
                continue
            result['vels'].append(math.sqrt((result['reach_dist_x'][i]-
result['reach_dist_x'][i-1])*(result['reach_dist_x'][i]-result['reach_dist_x']
[i-1])+(result['reach_dist_y'][i]-result['reach_dist_y'][i-1])*
(result['reach_dist_y'][i]-result['reach_dist_y'][i-1])/0.004))
```

2.绘制神经元raster和PSTH图

Raster plots可以用于分析神经元在不同刺激或条件下的活动模式。PSTH即刺激时间周围的时间直方图，以显示神经元在刺激条件下的活动模式。在本次实验中，我观察了方向选择上的tuning，即当神经元对于特定的方向或运动方向具有更强的响应时。

raster中，我们根据输入的方向，在 reach_angle中找到对应方向的试验索引。然后使用这些索引筛选出对应的试验开始时间和结束时间。遍历每个选择的试验，提取 M1 脑电信号中的特定通道，根据试验开始和结束时间，截取对应的时间窗口。将截取的时间转换为相对试验开始时间的偏移，并将每个试验的 raster 数据添加到列表 raster中。在raster plot上，你可以观察到在特定方向上神经元活动更加显著的模式。

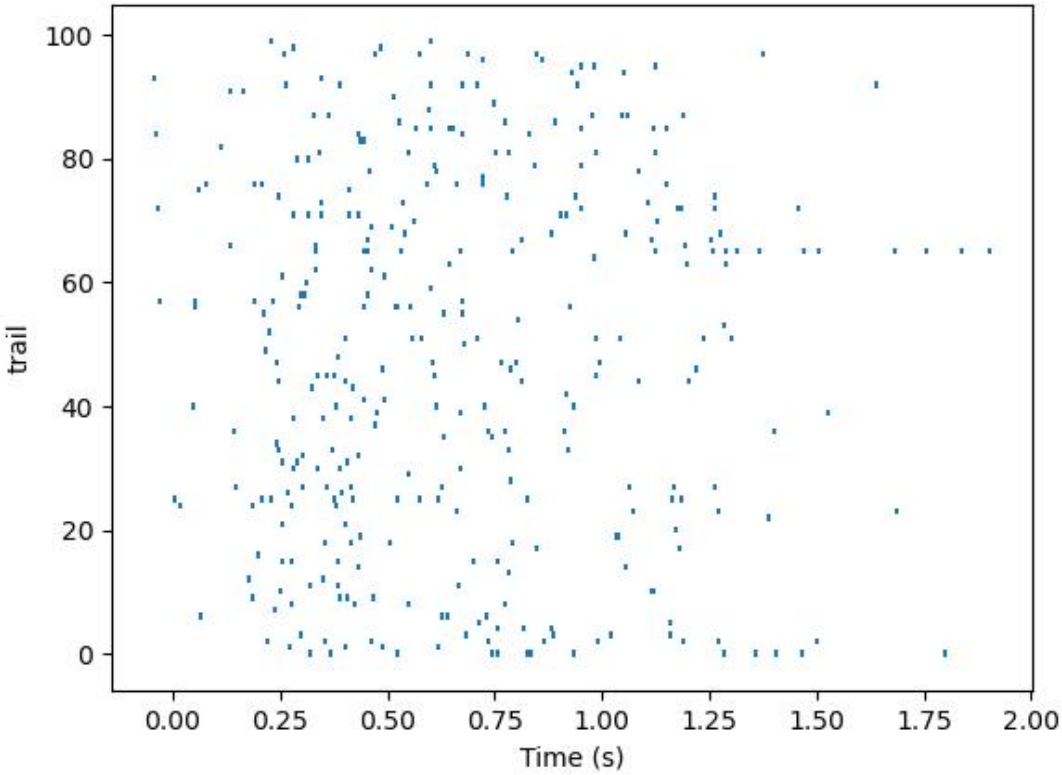
```
    for i in range(len(index[0])):
        temp_raster = X[0]
        start_timestamp = start_times[i] * 0.004 + t_min
        end_timestamp = end_times[i] * 0.004 + t_min
        temp_raster = temp_raster[start_timestamp - 20 *0.004 < temp_raster]
        temp_raster = temp_raster[temp_raster < end_timestamp]
        trans_raster = temp_raster - start_timestamp
        if trans_raster.shape[0] > 0:
            raster.append(trans_raster)
```

PSTH中，我们同样根据输入的方向，在 reach_angle 中找到对应方向的试验索引。然后使用这些索引筛选出对应的试验开始时间和结束时间。遍历每个选择的试验，提取 M1 脑电信号中的特定通道。根据试验开始和结束时间，截取对应的时间窗口，将神经元的 spike 时间转换为相对于试验开始时间的偏移。

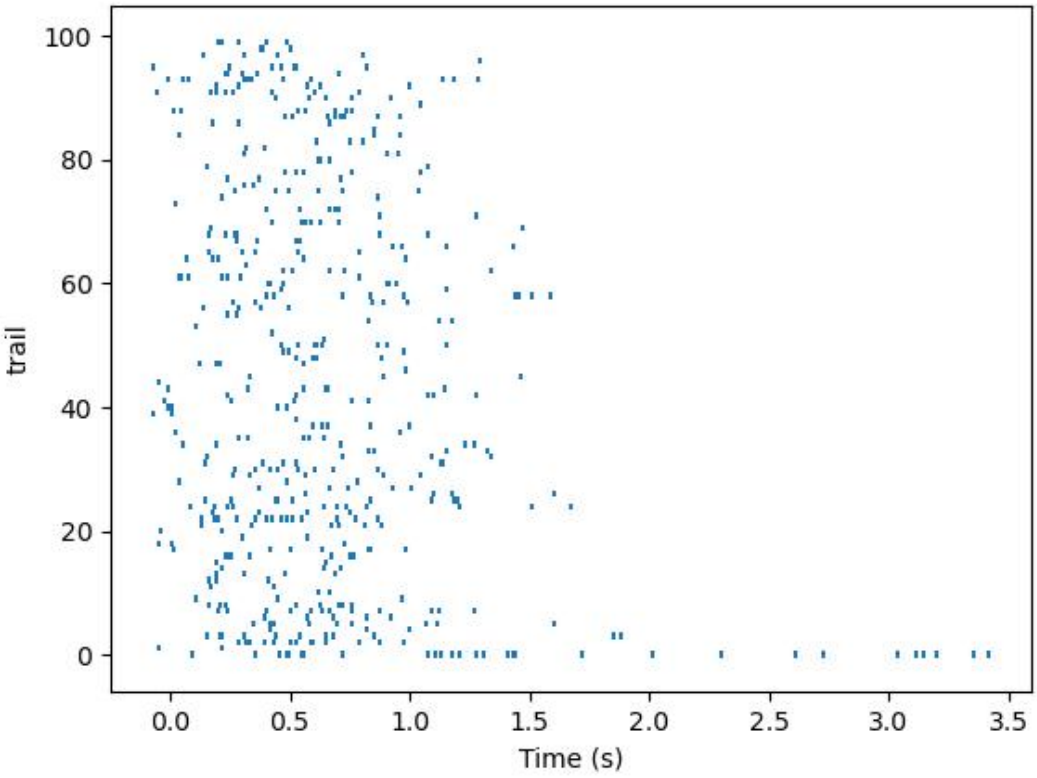
```
    for i in range(len(index[0])):
        temp_raster = X[0]
        start_timestamp = start_times[i] * 0.004 + t_min
        end_timestamp = end_times[i] * 0.004 + t_min
        temp_raster = temp_raster[start_timestamp - 20 *0.004 < temp_raster]
        temp_raster = temp_raster[temp_raster < end_timestamp]
        trans_raster = temp_raster - start_timestamp
        if trans_raster.shape[0] > 0:
            max = max if trans_raster[0].max() < max else trans_raster[0].max()
            min = min if trans_raster[0].min() > max else trans_raster[0].min()
            raster += trans_raster.tolist()

    n_bins = int(np.floor((max - min) / bin_width_s))
```

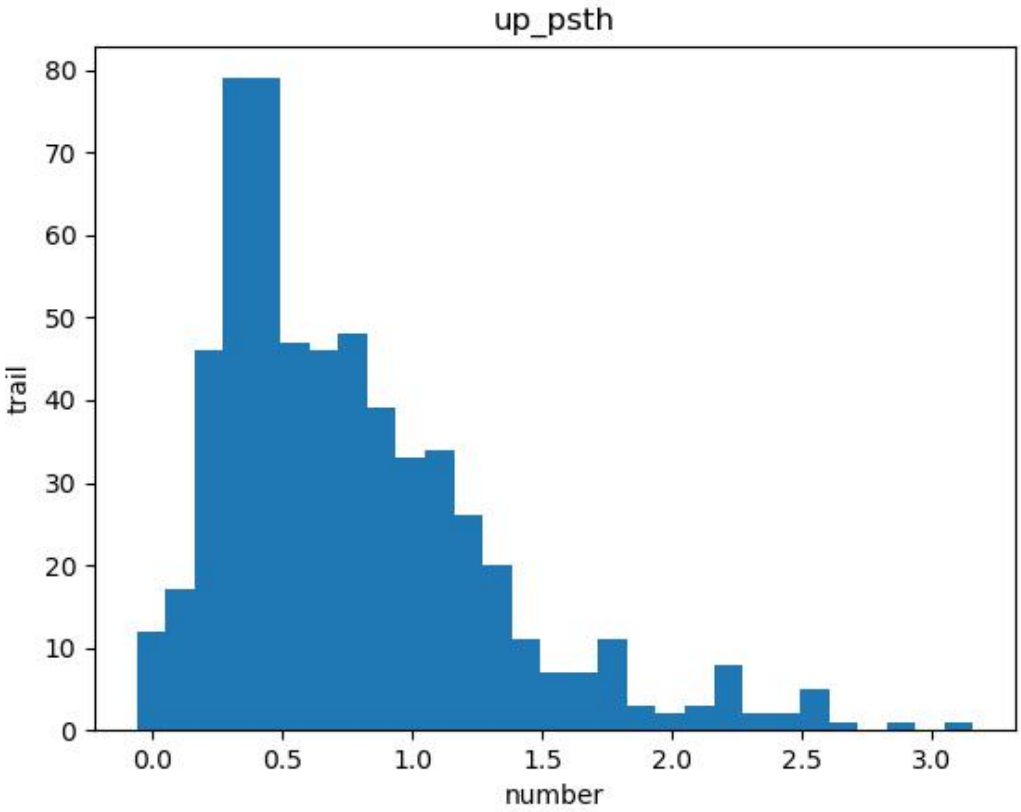
90°方向的raster plot：



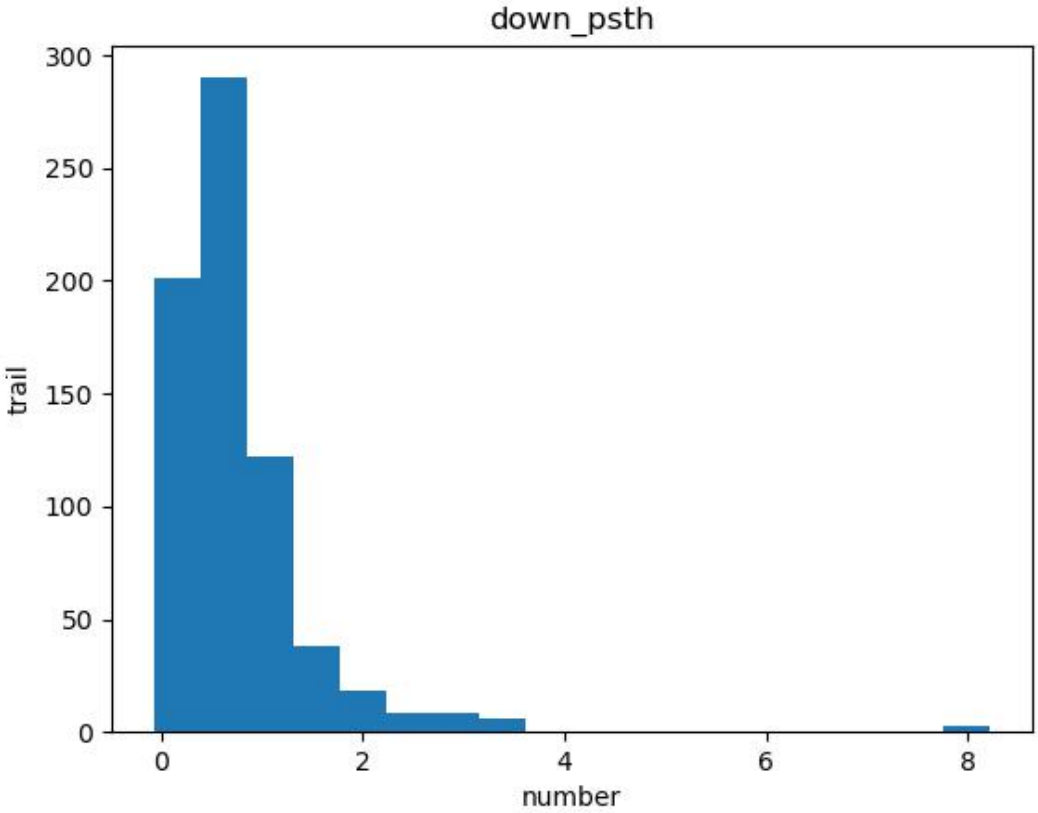
-90°方向的raster plot：



90°方向的PSTH图：



-90°方向的PSTH图：



3.绘制神经元的tuning curve

调谐曲线显示了神经元在不同刺激条件（这里是不同的角度）下的平均活动水平。

我们设置调谐曲线的角度区间，这里将整个360度的角度划分为12个区间，每个区间30度。使用循环遍历每个角度区间，根据角度区间选择对应的试验。在循环中，通过条件判断找到属于当前角度区间的试验索引，然后提取对应的试验开始时间和结束时间。对于每个试验，在对应的时间窗口内提取神经元的活动。这里使用了一个简单的时间窗口，根据试验的开始和结束时间截取神经元活动的片段，并将活动次数累积到 binned_spikes 数组中。

神经元tuning的运动参数有方向调谐、速度调谐、位置调谐等。

（1）方向调谐指的是 衡量神经元对于运动方向的选择性。神经元可能更喜欢特定方向的运动刺激，而对其他方向的刺激响应较弱。

（2）速度调谐表示神经元对于运动的速度的选择性。某些神经元可能对较慢或较快的运动刺激有更强烈的响应。

（3）位置调谐衡量神经元对于刺激在视网膜上位置的选择性。某些神经元可能对刺激在特定位置上的变化更为敏感。

（4）加速度调谐指的是神经元对于刺激的变化率的敏感性。在某些感觉系统中，对于运动加速度的敏感性可能是重要的。

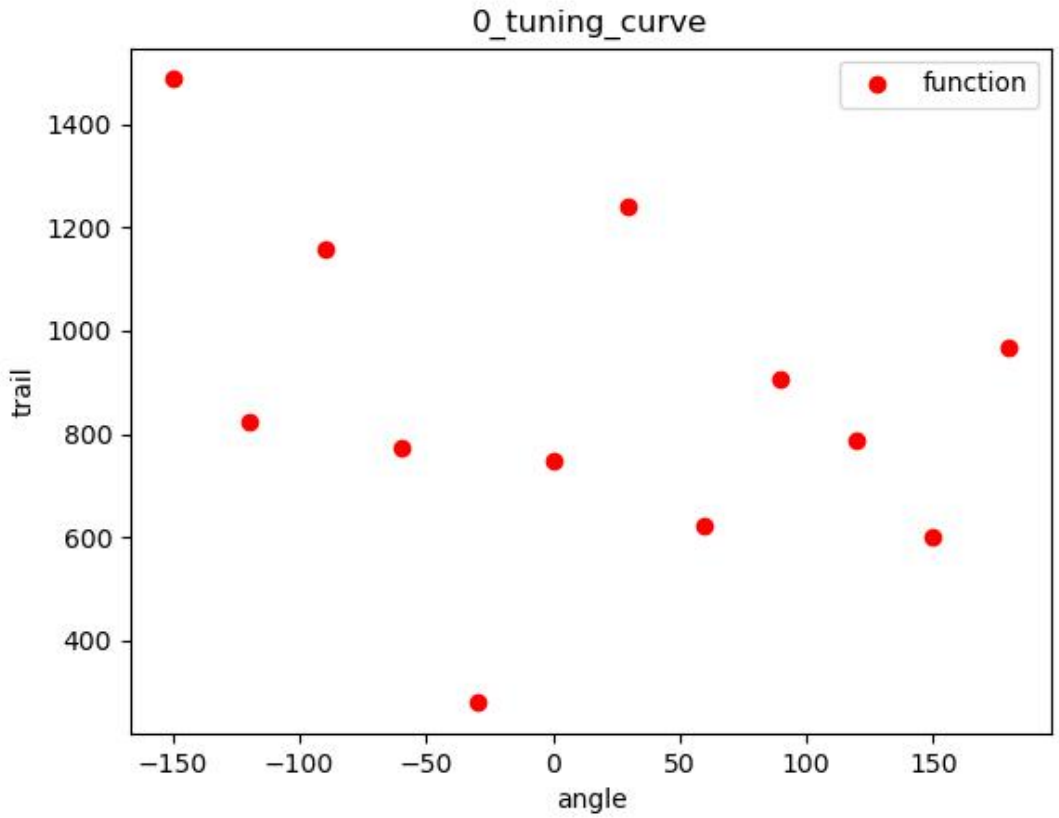
```
    for i in range(12):
        if i==6:
            start_angle1 = -15. + 6 * 30.
            end_angle1 = 15. - 6 * 30.
            index = np.where(((start_angle1 < reach_angle) & (180. >=
reach_angle)) | ((-180. < reach_angle) & (end_angle1 >= reach_angle)))
            x.append(i*30)
        elif i < 6:
            start_angle = -15. + i * 30.
```

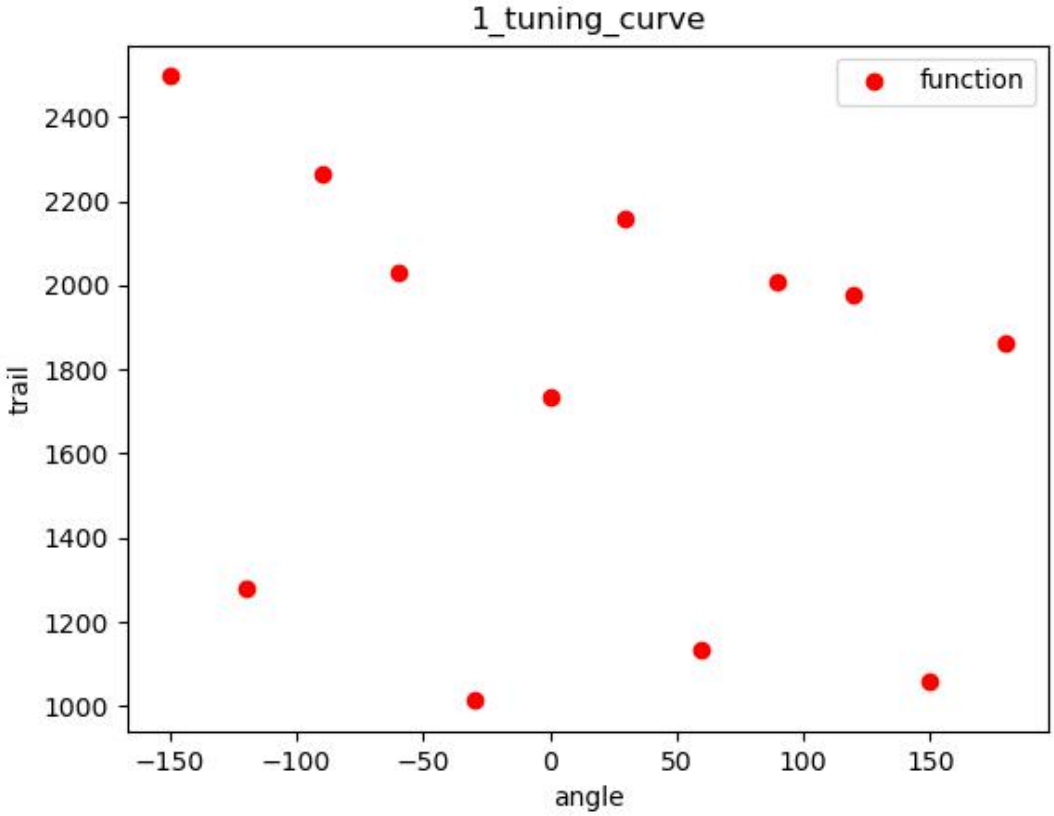
```
            end_angle = 15. + i * 30.
            index = np.where((start_angle < reach_angle) & (end_angle >=
reach_angle))
            x.append(i*30)
        else:
            start_angle = -360. + i * 30. - 15.
            end_angle = -360. + i * 30. + 15.
            index = np.where((start_angle < reach_angle) & (end_angle >=
reach_angle))
            x.append(i*30 - 360)

        start_time = start_times[index]
        end_time = end_times[index]

        raster = []
        for temp_i in range(len(index[0])):
            temp_raster = X[cell]
            start_timestamp = start_time[temp_i] * 0.004 + t_min
            end = end_time[temp_i] if (end_time[temp_i] - start_time[temp_i]) <
1000 else (1000 + start_time[temp_i])
            end_timestamp = end * 0.004 + t_min
            temp_raster = temp_raster[start_timestamp < temp_raster]
            temp_raster = temp_raster[temp_raster < end_timestamp]
            binned_spikes[i] += temp_raster.shape[0]
```

部分神经元的tuning curve图：





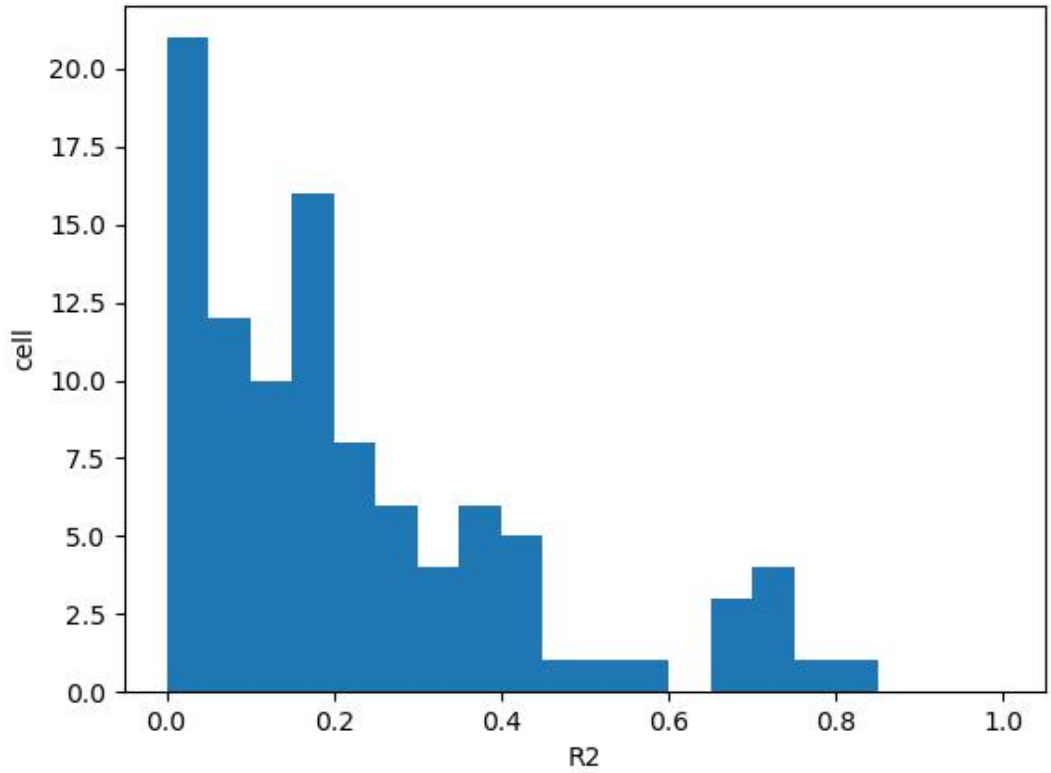
任务2 神经元运动调制分析

在本次实验中，我们选用R2作为模型对实际神经活动的解释程度的一个评估参数。我们将上图中的神经元tuning curve使用正弦函数模型进行拟合，对拟合结果计算R2，接着以0.05为区间计算各个区间内的神经元数量，并绘制图表。其中我们将角度分成12份，但(30*x*-*15*,*30*x+15)内的数据我们都归到30x这个点上，而后使用函数进行拟合。

代码中，`func` 函数定义了拟合的目标函数，其中 `a0` 是振幅，`a2` 是相位，`a3` 是纵向偏移。这里使用正弦函数，通过调整参数来拟合观察到的神经元活动。接着使用 `scipy.optimize.curve_fit` 函数进行拟合。`p0` 是拟合的初始参数。最后计算拟合的质量指标 R2，衡量拟合的好坏程度。

```
    y = binned_spikes[index]
    x = x[index]
    def target_func(x, a0, a2, a3):
        return a0 * np.sin((x + a2)/360*2*pi) + a3
    # a0*sin(a1*x+a2)+a3
    import scipy.optimize as optimize
    a0 = (max(y) - min(y)) / 2
    max_index = y.tolist().index(max(y))
    a2 = 90 - x[max_index]
    a3 = a0
    p0 = [a0, a2, a3]
    para, _ = optimize.curve_fit(target_func, x, y, p0=p0)
    print(para)
    y_fit = [target_func(a, *para) for a in x]
    #Get metric of fit
    y_mean=np.mean(y)
    R2=1-np.sum((y_fit-y)**2)/np.sum((y-y_mean)**2)
    print('R2s:', R2)
```

最终R2绘制的直方图：



任务3 实战：基于卡尔曼滤波器的运动解码

此部分基于老师钉钉群里的kalman滤波器链接：https://github.com/KordingLab/Neural_Decoding。

kalman滤波器：

```
class KalmanFilterRegression(object):

    """
    Class for the Kalman Filter Decoder

    Parameters
    ----------
    C - float, optional, default 1
    This parameter scales the noise matrix associated with the transition in
kinematic states.
    It effectively allows changing the weight of the new neural evidence in the
current update.

    Our implementation of the Kalman filter for neural decoding is based on that
of Wu et al 2003 (https://papers.nips.cc/paper/2178-neural-decoding-of-cursor-
motion-using-a-kalman-filter.pdf)
    with the exception of the addition of the parameter C.
    The original implementation has previously been coded in Matlab by Dan Morris
(http://dmorris.net/projects/neural_decoding.html#code)
    """

    def __init__(self,C=1):
        self.C=C


    def fit(self,X_kf_train,y_train):
```

```
        """
        Train Kalman Filter Decoder

        Parameters
        ----------
        X_kf_train: numpy 2d array of shape [n_samples(i.e. timebins) ,
n_neurons]
            This is the neural data in Kalman filter format.
            See example file for an example of how to format the neural data
correctly

        y_train: numpy 2d array of shape [n_samples(i.e. timebins), n_outputs]
            This is the outputs that are being predicted
        """

        #First we'll rename and reformat the variables to be in a more standard
kalman filter nomenclature (specifically that from Wu et al, 2003):
        #xs are the state (here, the variable we're predicting, i.e. y_train)
        #zs are the observed variable (neural data here, i.e. X_kf_train)
        X=np.matrix(y_train.T)
        Z=np.matrix(X_kf_train.T)

        #number of time bins
        nt=X.shape[1]

        #Calculate the transition matrix (from x_t to x_t+1) using least-
squares, and compute its covariance
        #In our case, this is the transition from one kinematic state to the
next
        X2 = X[:,1:]
        X1 = X[:,0:nt-1]
        A=X2*X1.T*inv(X1*X1.T) #Transition matrix
        W=(X2-A*X1)*(X2-A*X1).T/(nt-1)/self.C #Covariance of transition matrix.
Note we divide by nt-1 since only nt-1 points were used in the computation
(that's the length of X1 and X2). We also introduce the extra parameter C here.

        #Calculate the measurement matrix (from x_t to z_t) using least-squares,
and compute its covariance
        #In our case, this is the transformation from kinematics to spikes
        H = Z*X.T*(inv(X*X.T)) #Measurement matrix
        Q = ((Z - H*X)*((Z - H*X).T)) / nt #Covariance of measurement matrix
        params=[A,W,H,Q]
        self.model=params

    def predict(self,X_kf_test,y_test):

        """
        Predict outcomes using trained Kalman Filter Decoder

        Parameters
        ----------
        X_kf_test: numpy 2d array of shape [n_samples(i.e. timebins) ,
n_neurons]
            This is the neural data in Kalman filter format.

        y_test: numpy 2d array of shape [n_samples(i.e. timebins),n_outputs]
            The actual outputs
```

```
            This parameter is necesary for the Kalman filter (unlike other
decoders)
            because the first value is nececessary for initialization

        Returns
        -------
        y_test_predicted: numpy 2d array of shape [n_samples(i.e.
timebins),n_outputs]
            The predicted outputs
        """

        #Extract parameters
        A,W,H,Q=self.model

        #First we'll rename and reformat the variables to be in a more standard
kalman filter nomenclature (specifically that from Wu et al):
        #xs are the state (here, the variable we're predicting, i.e. y_train)
        #zs are the observed variable (neural data here, i.e. X_kf_train)
        X=np.matrix(y_test.T)
        Z=np.matrix(X_kf_test.T)

        #Initializations
        num_states=X.shape[0] #Dimensionality of the state
        states=np.empty(X.shape) #Keep track of states over time (states is what
will be returned as y_test_predicted)
        P_m=np.matrix(np.zeros([num_states,num_states]))
        P=np.matrix(np.zeros([num_states,num_states]))
        state=X[:,0] #Initial state
        states[:,0]=np.copy(np.squeeze(state))

        #Get predicted state for every time bin
        for t in range(X.shape[1]-1):
            #Do first part of state update - based on transition matrix
            P_m=A*P*A.T+W
            state_m=A*state

            #Do second part of state update - based on measurement matrix
            K=P_m*H.T*inv(H*P_m*H.T+Q) #Calculate Kalman gain
            P=(np.matrix(np.eye(num_states))-K*H)*P_m
            state=state_m+K*(Z[:,t+1]-H*state_m)
            states[:,t+1]=np.squeeze(state) #Record state at the timestep
        y_test_predicted=states.T
        return y_test_predicted

KalmanFilterDecoder = KalmanFilterRegression
```

接着使用kalman滤波器进行拟合预测：

```
    model_kf=KalmanFilterDecoder(C=1) #There is one optional parameter that is
set to the default in this example (see ReadMe)

    #Fit model
    model_kf.fit(X_kf_train,y_kf_train)

    #Get predictions
    y_valid_predicted_kf=model_kf.predict(X_kf_valid,y_kf_valid)
```

Q1 PQR设置

Q：过程激励噪声的协方差矩阵。R：观测噪声的协方差矩阵。 P：不断迭代计算的估计误差的协方差矩阵。

P矩阵表示状态估计的协方差矩阵，它描述了状态估计和真实状态之间的不确定性。P矩阵的初始值可以根据系统的初始状态估计精度进行设置。本实验中P的初始值为零矩阵，表示对初始状态估计的高置信度。

Q矩阵表示系统模型中过程噪声的协方差矩阵。它描述了系统状态在时间上的变化和不确定性。通常情况下，Q矩阵的初始值可以根据系统的动态范围和预期的噪声水平进行估计。本实验中Q初始通过计算状态转移矩阵的最小二乘解并计算其协方差得到。

R矩阵表示测量模型中观测噪声的协方差矩阵。它描述了观测值和系统真实状态之间的不一致性或不确定性。R矩阵的初始值可以通过对测量数据进行统计分析来估计。本实验中R初始通过计算状态转移矩阵的最小二乘解并计算其协方差得到。

Q2 PQR初始值影响

P表示我们对当前预测状态的信任度，它越小说明我们越相信当前预测状态；它的值决定了初始收敛速度，一般开始设一个较小的值以便于获取较快的收敛速度。随着卡尔曼滤波的迭代，P的值会不断的改变，当系统进入稳态之后P值会收敛成一个最小的估计方差矩阵，这个时候的卡尔曼增益也是最优的，所以这个值只是影响初始收敛速度。

Q值为过程噪声，越小系统越容易收敛，我们对模型预测的值信任度越高；但是太小则容易发散。如果Q为零，那么我们只相信预测值；Q值越大我们对于预测的信任度就越低，而对测量值的信任度就变高；如果Q值无穷大，那么我们只信任测量值。

R值为测量噪声。`R`太大，卡尔曼滤波响应会变慢，因为它对新测量的值的信任度降低；越小系统收敛越快，但过小则容易出现震荡。

Q3 不同实验参数对结果的影响

	only pos	only vel	only acc	pos&vel	all
R2	0.3516	0.2128	0.0484	0.3854 0.4893	0.4284 0.3096 0.0146
rho2	0.3699	0.3059	0.0713	0.4677 0.5096	0.5056 0.3511 0.0815

从表中可以看到当参数越多，对位置的拟合程度越高。当只有位置和速度两个参数时，对速度拟合程度最高。而对加速度，拟合效果均不太理想。

Q4 不同方法对比

本实验统一使用实验参数为位置和速度，拟合程度使用速度的R2作为标准。

```
    #lstm
    model_lstm=LSTMRegression(units=512,dropout=0.1,num_epochs=10)

    #Fit model
    model_lstm.fit(X_train,y_train)

    #Get predictions
    y_valid_predicted_lstm=model_lstm.predict(X_valid)

    #Get metric of fit
    R2s_lstm=get_R2(y_valid,y_valid_predicted_lstm)
    print('R2s:', R2s_lstm)
```

```
    #linear
    model_linear=linear_model.LinearRegression()

    #Fit model
    model_linear.fit(X_train,y_train)

    #Get predictions
    y_valid_predicted_linear=model_linear.predict(X_valid)

    #Get metric of fit
    R2s_linear=get_R2(y_valid,y_valid_predicted_linear)
    print('R2s:', R2s_linear)
```

	kf	lstm	linear
R2 of vel	0.2974	0.4086	0.6291

实验中，卡尔曼滤波器效果不佳的原因，我认为可能在于Q、R的设置不合理导致的。

任务4 结论梳理

本次实验，我们首先进行了数据处理，筛选出实验需要的数据，接着对数据进行了raster和PSTH的展示，同时绘制了部分神经元的tuning curve图。根据tuning curve图，我们使用正弦函数进行拟合，并对拟合程度进行可视化展示。最后使用卡尔曼滤波器对神经元的运动状态进行拟合，并进行了对比实验。

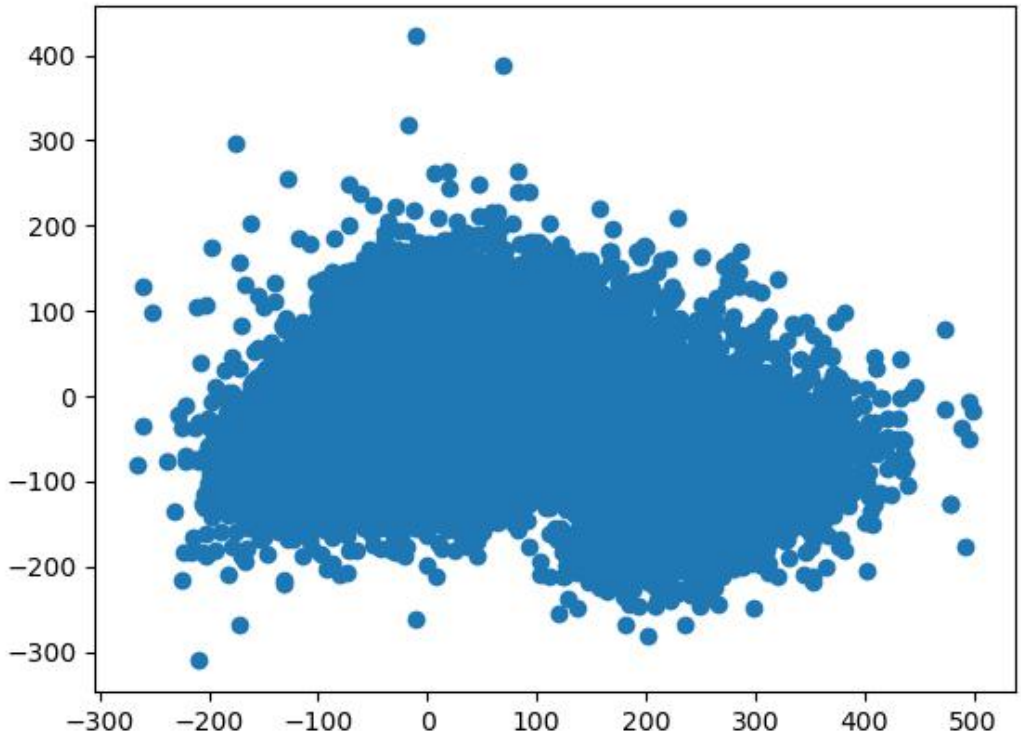
本次实验，我们在任务二中使用不同神经元的tuning curve进行正弦函数的拟合，从结果上看，大部分数据在正弦函数的拟合下效果并不出色。一方面可能是区间划分的不够细，另一方面可能是因为函数选择的不好，二者都导致拟合效果不佳。在任务三中，我们使用了卡尔曼滤波器对神经元的运动状态进行拟合，并对输入参数的不同依次进行了实验，其中加速度的拟合程度不佳；而参数越多时，对位置的拟合程度越好，但参数只有位置和速度时，速度拟合程度最好。最后我们将其与lstm和线性拟合进行对比，效果不如二者。我认为可能原因是由于个人对这方面不熟悉的原因，导致Q、R的参数设置不合理导致的。

附加题 神经元锋电位分类

任务1 神经元锋电位波形可视化和分析

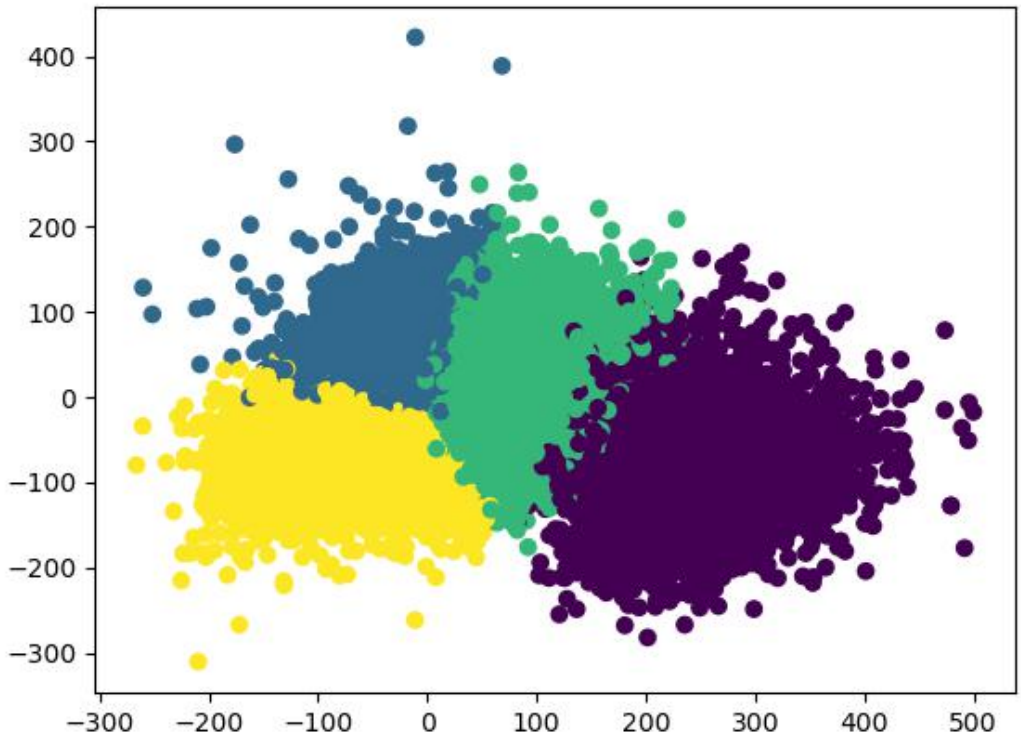
本次实验随机使用22号通道来进行实验。首先从数据中提取未排序的脑电波形信号，接着以8个主成分数量来建立PCA模型，对波形信号进行 PCA 变换，得到新的特征矩阵，该矩阵包含了原始信号中最重要的8个主成分。

```
        n_sorted_units = f["wf"].shape[0] - 1  # The FIRST one is the 'hash' --
ignore!
        unsorted_spike = f[f["wf"][0, chan_idx]][()].T
        pca_sk = PCA(n_components=8)
        newMat = pca_sk.fit_transform(unsorted_spike)
```



使用K-means进行聚类：

```
        n_clusters = 4
        cluster = KMeans(n_clusters=n_clusters,random_state=0).fit(newMat)
```



任务2 分类结果评估和分析

比较

我们使用基于 Gini 指数的群集质量度量，用于衡量在 PCA 空间中对未排序的脑电信号进行聚类的效果。

1. 遍历每个单元：
   a. 从数据中提取单元的脑电波形信号 。
   b. 对脑电波形信号进行 PCA 变换。
   c. 使用已经训练好的聚类模型 `cluster` 对 PCA 变换后的数据进行聚类。
   d. 计算聚类结果的 Gini 指数。
2. 计算每个单元的长度在整个数据集中所占比例。
3. 输出总的 Gini 指数，即对所有单元的 Gini 指数进行加权求和

最终得到的Gini指数为0.3719，当以wf2-5列为round truth时，gini指数为0.3562，实验模型性能略低于ground truth。

现有问题

（1）PCA通过保留数据中的主要方差来实现降维，但这可能导致一些次要但有用的信息被舍弃。在某些情况下，降维可能导致对数据结构的损失。

（2） PCA假设数据是线性可分的，即主成分是数据中的线性组合。对于非线性结构的数据，PCA的表现可能不佳。

举例：

（1）不同类型的神经元在某些特征上可能存在重叠，使得PCA模型难以准确地将它们区分开来进行降维。这可能是因为特征选择不足或类别之间存在较大的变异性。

（2）神经元的类型与它们在空间上的分布有关，并不是线性可分的，而PCA模型未能充分考虑空间结构，可能导致分类性能下降。

任务3 结论梳理

此次实验我们使用PCA对脑电波形信号进行降维和可视化，接着使用k-means方法对降维后的数据进行聚类和可视化。最后我们设计了基于Gini指数的集群聚类的度量标准，与ground truth进行了对比，并对结果进行了分析，对目前存在的问题进行了讨论。

实验首先进行了降维和分类，生成了相应的图。接着使用Gini指数来衡量PCA聚类效果。最终得到的Gini指数为0.3719，当以wf2-5列为round truth时，gini指数为0.3562，实验模型性能略低于ground truth。目前一方面，不同类型的神经元在某些特征上可能存在重叠，PCA通过保留数据中的主要方差来实现降维，但这可能导致一些次要但有用的信息被舍弃。在某些情况下，降维可能导致对数据结构的损失。另一方面，神经元的类型与它们在空间上的分布有关，并非是简单的线性可分结构，而PCA模型未能充分考虑空间结构，可能导致分类性能下降。