问题1

模型分析，基于以下网络我们探究CMSA机理

```
class Network:
    def __init__(self, En, In, rp, We, Wi) -> None:
        self.E_neurons = LIFlayer(n=En,refractory_period=rp)
        self.synapsesEE = Synapseslayer(En,En,101,W=We)
        self.synapsesEI = Synapseslayer(En,In,101,W=We)

        self.I_neurons = LIFlayer(n=In, refractory_period=rp)
        self.synapsesIE = Synapseslayer(In,En,101,Vrest=-80,sigma=400,W=Wi)
        self.synapsesII = Synapseslayer(In,In,101,Vrest=-80,sigma=400,W=Wi)

    def update(self, inputE:torch.Tensor, inputI:torch.Tensor):
        E_potential, E_spike = self.E_neurons.update(inputE+self.synapsesEE.i.sum(dim=(2,3))+sel
        I_potential, I_spike = self.I_neurons.update(inputI+self.synapsesII.i.sum(dim=(2,3))+sel
        self.synapsesEE.update(E_spike, E_potential)
        self.synapsesEI.update(E_spike, I_potential)
        self.synapsesIE.update(I_spike, E_potential)
        self.synapsesII.update(I_spike, I_potential)

        return E_potential,E_spike
```

在网络中我们有两个LIF层 E_neuron, I_neuron，分别为抑制和激发，同时网络中存在相互作用机理，模型中通过抑制作用 $GABA$ 受体突触实现，激发作用使用 $AMPA$ 受体，实现E层激发自身和I层，I层抑制自身和E层，分别探究了在不同参数下刺激网络后 E_neuron 层的膜电位变化，通过调整参数复现了自发态(bump)，激发态(wave)和临界态(critical state)。以下是一些具体的实现过程和一些现象原因分析。

LIF神经元的update函数

我们通过以下代码建模了LIF神经元的动力学模型

```
class LIFlayer:
    def __init__(self, n:int, threshold=-50.0, reset_value=-70.0, membrane_capacitance=1.0, gL=0

    def update(self, input:torch.Tensor):
        assert input.shape == self.shape
        #Todu 请你完成膜电位、神经元发放和不应期的更新
        global __t__ ,dt
        #计算变化量
        dV_dt=(-self.gL*(self.potential-torch.full(self.shape,self.reset_value).to(device))+inpu
        #计算变化后的电位
        self.potential=self.potential+dV_dt*dt
        #是否spike
        self.spike = (self.potential > self.threshold).bool() * (__t__-self.spike_time-self.refr
        # 超过阈值且不在不应期内才发放
        #更新兴奋神经元的发放时间
        self.spike_time[self.spike]= __t__
        #将不应期的神经元设置为静息电位
        self.potential[(__t__-self.spike_time<self.refractory).bool()]=self.reset_value
        return self.potential, self.spike
```

根据基本的泄漏整合发放神经元模型：

$$C\frac{dV}{dt} = -g_L * (V - V_{rest}) + I_{input}$$

相比于 HH 模型这个模型更加简便，更适合网络层面的计算。
我们在每一个时间间隔更新每个神经元的膜电位，并判断是否会产生spike，如果电位超过阈值并且发放时间间隔超过不应期（不应期是我们后续分析原因和调参的重要指标），并且更新spike后的神经元的 `spike_time`，使其进入不应期。

Synapse层的update函数

突触层分为抑制型层和兴奋型层，分别模拟GABA和AMPA两种突触，我们修改的代码如下：

```
def gaussian(self, n, W, sigma):
    #Tosu 请你完成高斯波包函数，返回一个n*n矩阵，其中最大值位于正中间（n为奇数）
    # 计算中心点位置
    center = (n - 1) / 2
    # 生成坐标网格
    x = torch.arange(n, device=device)
    y = torch.arange(n, device=device)
    X, Y = torch.meshgrid(x, y, indexing='ij')  # 使用 'ij' 索引方式，确保行列顺序正确
    # 计算每个位置到中心点的平方距离
    distance_sq = (X - center) ** 2 + (Y - center) ** 2

    # 计算高斯波包矩阵
    gaussian = W * torch.exp(-distance_sq / sigma)
    '''
    # 转换为 numpy 数组
    numpy_matrix = gaussian.clone().cpu().numpy()
    # 保存到文本文件
    np.savetxt(f'matrix{W}.txt', numpy_matrix, fmt='%.4f')
    '''
    return gaussian

def update(self, input: torch.Tensor, potential:torch.Tensor):
    assert input.shape == (self.in_neurons, self.in_neurons)
    # change to matrix size to be adaptive
    if self.in_neurons<self.out_neurons:
        input = self.scale_up(input,self.out_neurons//self.in_neurons)
    else:
        input = self.scale_down(input,self.in_neurons//self.out_neurons)
    global dt ,__t__
    #更新电导
    dg_dt=(-self.g + torch.einsum('abcd,cd->abcd',input, self.weight))/self.time_constant
    self.g=self.g+dt*dg_dt
    #展开膜电位并计算电流
    expanded_potential = potential.unsqueeze(2).unsqueeze(3).repeat(1, 1, self.shape[2], sel
    self.i=self.g*(self.Vrest-expanded_potential)
    return self.i
```

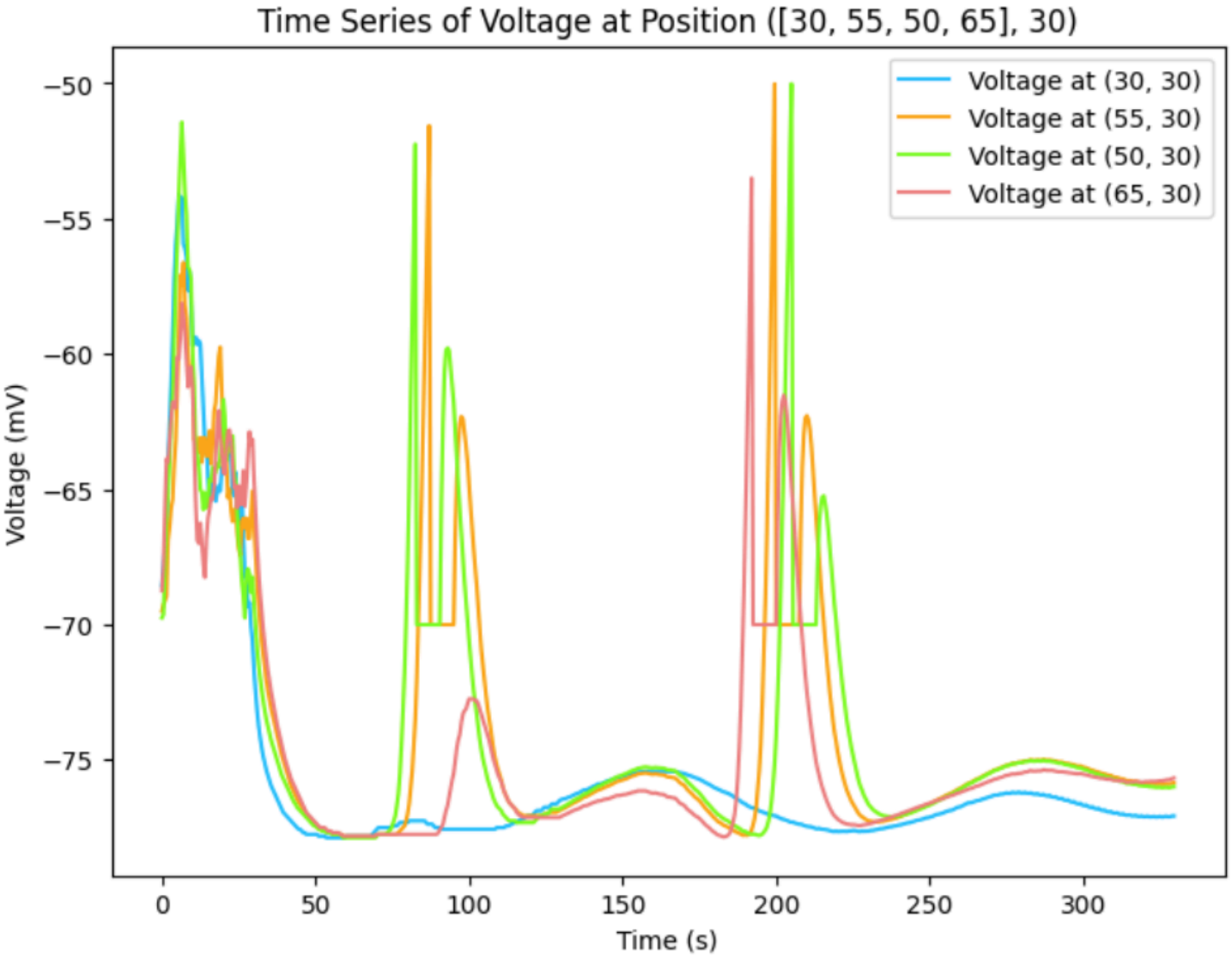
这里我们采用高斯样的兴奋耦合和抑制耦合，通过 `gaussian(self, n, W, sigma)` 函数计算突触的连接权重，

$$W_{ij} = W \cdot e^{-\frac{(x_i-x_j)^2+(y_i-y_j)^2}{\sigma}}$$

W_{ij} 表示为神经元 i 对神经元 j 的兴奋（抑制）强度。
具体调参时通过更改W的值来控制兴奋性突触和抑制性突触的连接强度。

模拟参数的设置摸索

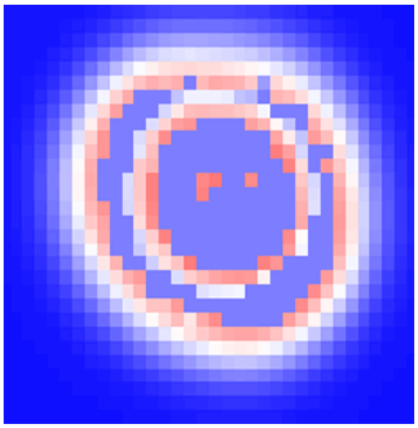
在真正意义上的调参从而观察到网络状态的变化之前，我们首先进行了大量的debug。
最终发现问题在于将 W_i 设置成了负值，这相当于整个网络不断在刺激，从而膜电位在外界刺激时连续震荡，在撤去外界输入后整体兴奋或抑制，无法产生连续而稳定的波包。如图是正确的网络中几个神经元的膜电位变化。



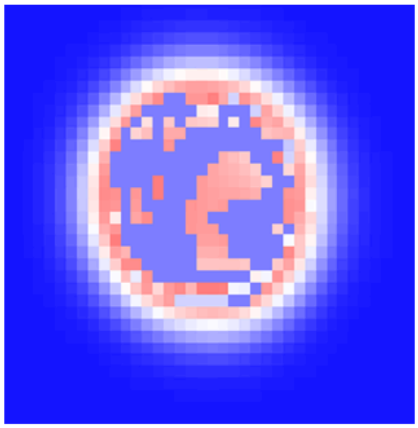
由于超参数 W_e, W_i, rp （激发强度、抑制强度、不应期）共同作用于波包的状态，通过观察我们发现当W_e和W_i相对合理（指自发态能产生形状稳定，数目约5-6，spike发放频率稳定的波包）的情况下($W_e = 0.12, W_i = 0.06$)，我们仅仅更改不应期就可以看到由bump到critical state再到wave状态的稳定转变。所以以下我们讨论在确定的W_e和W_i下改变不应期波包的形态变化。根据实验结果当 $5ms \leq rp \leq 5.5ms$ 时是标准的自发态，当 $6 \leq rp \leq 7.2$ 为临界态，当 $7.5 \leq rp \leq 8.5$可以观察到行波。

波包的形成分析

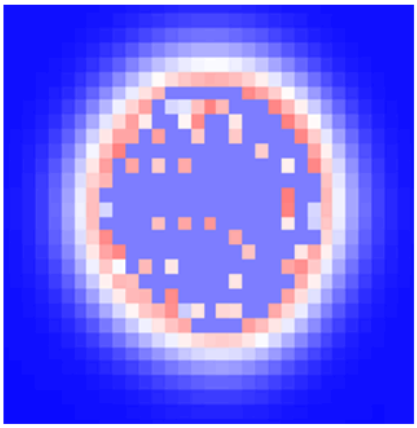
在自发态，我们仔细观察了 bump 波包spike的发放模式，具体有三种：明显的由内向外的环状spike（如图a）；spike频率非常高而形成的波包内闪烁（如图b）；波包内部spike的神经元相对分散（如图c）：



a. 由中心环状向外扩散的spike



b. spike频率过高，环状spike呈现为波包内的闪烁



c. 没有环状的spike，波包内部spike的神经元较分散

我们推测，波包产生是由于某一区域的神经元膜电位在附近其他神经元的通过突触的激发和抑制作用下达到相对平衡的状态，波包内部的神经元受到足以产生spike的兴奋性突触输入（强于抑制性），所以在spike后进入不应期，当不应期结束后进行下一次发放。而波包边缘的神经元由图中可以看到明显的由蓝（-80mV）到红（-60mV）的电位渐进性变化，受到的抑制较明显，保持一定的电位（静息电位和发放电位之间），这样有利于维持波包系统的能量，避免外部神经元发放导致的能量散失。

在调参过程中我们观察到：如果 W_e 保持不变减小 W_i 波包的直径会略微增大，推测相比于较大的 W_i 环状spike扩散至更远才收到足够的抑制。

如果同时增大 W_e 和 W_i 波包的发放频率会增大，推测系统激发和抑制作用共同增大，系统的能量更高，波包能维持形态（不至于因为兴奋过强而消失）同时脱离不应期后立即兴奋。当频率大于一定值时原本的环状扩散spike,变成波包内部神经元的闪烁。

临界态的出现

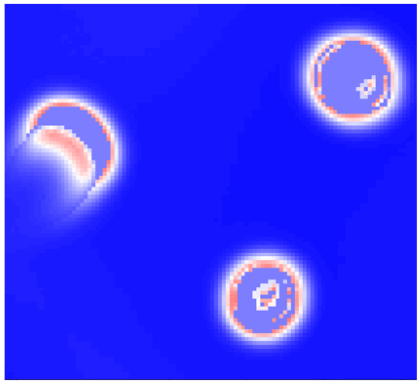
我们发现在自发态每个波包的形状基本呈现圆形，且大小和位置比较稳定，但是临界态我们可以同时观察到圆形和月牙形的波包，可以观察到二者的转化、椭圆形波包、椭圆形波包的内部spike特征、椭圆形波包在某一范围内扰动。

由自发态转变为临界态甚至激发态（产生速度恒定的行波）涉及此动力学系统的相变过程，具体的分析参见计算序参量部分的详细解释，这里直观和形象地推测可能的原因。

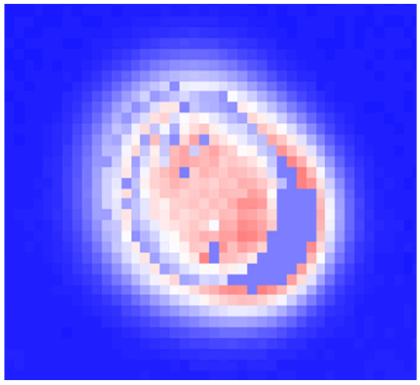
波包的 bump 时的spike不是完全从圆心发放且不是完全对称的。实际上存在噪声，但通常噪声相对微弱，不足以对波包的对称性造成影响，但是当我们增大 rp ，这种噪声被放大。当波包的中心向一侧发生 Δ 扰动时（即这一侧的神经元相比于另一侧提前 Δt 发放），如果不应期较小另一侧神经元spike后误差被弥补。如果不应期较大，可能在下一次发放时 Δt 增大，一定时间累计后，对称性破缺， bump 状态转变为 wave。而 critical stste 是在 rp 介于两者之间观察到的，体现为：1. 在某一时刻一部分波包

为行波，一部分波包在原地。2. 某一个波包在一定时段呈现为 wave， 一定时段呈现为 bump，一定时段以椭圆形的形态在一定范围内震荡，如图b。

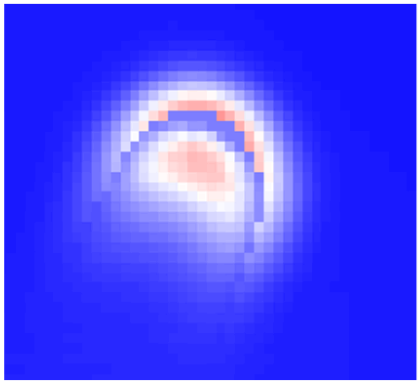
如图是临界态的特征：



a. Critical state，同时存在 wave 和 bump



b. Critical state 波包改变运动方向

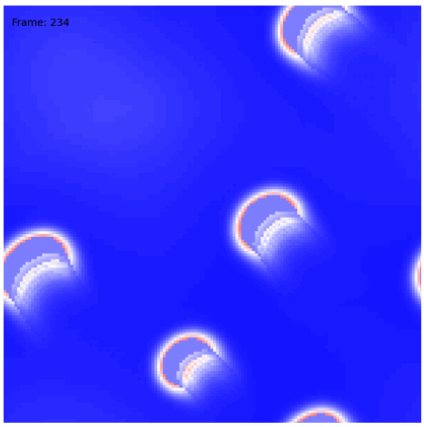


c. 临界态波包在运动过程中消失

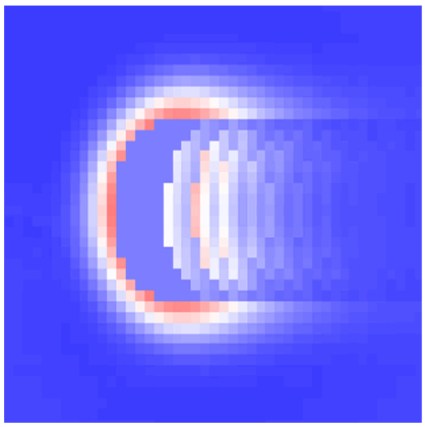
临界态波包不稳定，有可能在行波阶段发放减弱而消失。我们观察到相比于wave阶段的行波，临界态即将消失的行波“月牙形”更窄，即外侧spike和内侧spike的两部分神经元之间间隔的不应期神经元很少，这样不易维持平衡，外侧受到的抑制更大、前进的速度慢，内侧速度相对较快，当内侧spike后进入不应期而外侧还未从不应期中恢复，行波无法维持。

行波状态分析

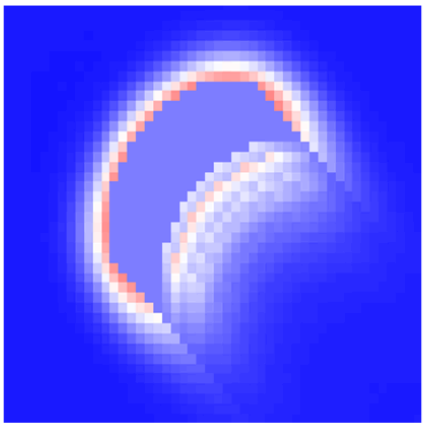
当我们继续增大不应期（约为8sm），可以观察到明显的行波。如图a：



a. 较好情况下wave方向一致



b. 月牙形行波弧度较大，轨迹残影更长



c. 抑制较弱时，行波两端长度大

当参数设置不合理（如 W_e 和 W_i 的比例过大），产生的行波没有一致的方向，我们在探索过程中观察到一些有趣的现象：1. 两个行波运动方向不平行，导致距离减小，产生相互作用，距离足够近时，出现一种类似排斥的效果，两个行波改变方向，彼此远离。2. 两个行波相向而行，发生“碰撞”，形成一个波包，停留在原地，出现 bump 的现象，可以推测碰撞时相接处的神经元spike后进入不应期使两个波无法继续向前传递。

当我们增大 rp 或者减小 W_i 发现月牙形的行波弧度变小，长度变大。推测抑制减小导致行波侧面的更容易spike，发放的速率与行波顶点（运动时最前方的点）速率差距变小，月牙形行波有被“拉开”的趋势。
但是当我们继续增大rp，例如增大到9，会出现不理想的结果，行波形态失去稳定性，从月牙形转变为长条形。这一现象当我们将W_i调整到0.2（很小，保持$W_e = 0.12, rp = 8$）也可以观察到，原因分析：不应期过长，外侧神经元受到的抑制更小，行波的能量向两侧扩散，导致行波无法维持稳定的状态。

问题2

计算序参量

计算每帧图的center list

首先，我们需要识别一张图中的所有pattern，而一个pattern的center定义为

$$I_{\mathrm{M}}(t) = \frac{1}{M_{\mathrm{f}}} \sum_{L=1}^{M_{\mathrm{f}}} \mathrm{i}_{\mathrm{M}}^{L}(t), J_{\mathrm{M}}(t) = \frac{1}{M_{\mathrm{f}}} \sum_{L=1}^{M_{\mathrm{f}}} \mathrm{j}_{\mathrm{M}}^{L}(t),$$

其中$\mathrm{i}_{\mathrm{M}}^{L}(t)$指的是时间t第M个pattern中第L个neuron的横坐标，$\mathrm{j}_{\mathrm{M}}^{L}(t)$的定义同理。

为了搜索pattern，我们使用flood-fill算法，也即对于一个发放的神经元，我们搜索它的周围，并把足够近的同样在发放的神经元视为同一个pattern。当在规定的邻近范围内找不到其它发放神经元时，我们记录搜索过的neuron并保存当前找到的pattern的编号和神经元列表。这部分工作由detect_patterns和flood_fill_pattern两个函数完成：（前者处理整个一帧图像，后者寻找一个pattern）

```
def detect_patterns(data, threshold=-53) -> list:#寻找一帧图像的pattern
    # 二值化数据
    binary = (data >= threshold).astype(int)
    height, width = binary.shape
    visited = np.zeros_like(binary, dtype=bool)
    patterns = []

    # 遍历所有神经元
    for i in range(height):
        for j in range(width):
            # 如果找到未访问的活跃神经元，开始flood fill
            if binary[i,j] == 1 and not visited[i,j]:
                pattern = flood_fill_pattern(binary, (i,j), visited)

                # 只保留大于最小大小的pattern
                if len(pattern) >= 5:  # 最小pattern大小，可调整
                    # 创建pattern掩码用于计算Euler特征
                    mask = np.zeros_like(binary)
                    for pi, pj in pattern:
                        mask[pi, pj] = 1

                    euler_number = calculate_euler_number(mask)
                    patterns.append({
                        'coords': pattern,
                        'type': 'crescent' if euler_number == 1 else 'patchy',
                        'euler': euler_number
                    })

    return patterns
```

然后再对于每一帧识别所有pattern并逐一计算center_list:

```
def calculate_centers_timeseries(data: np.ndarray) -> List[List[Tuple[float, float]]]:
    centers_timeseries = []
    #遍历所有帧
    for t in range(data.shape[0]):
        ...
```

计算velocities

为了计算Φ,我们需要先计算$\mathbf{v}$.

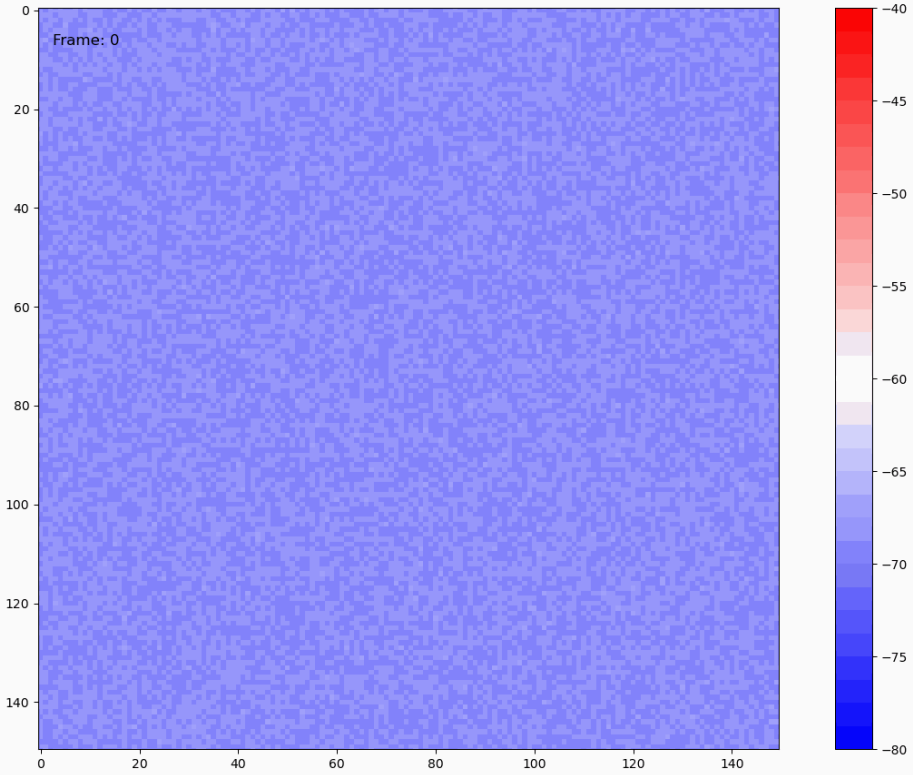
$$\Phi = \frac{\left\| \sum_{i=1}^{N} \overrightarrow{\mathbf{v}}_i \right\|}{\sum_{i=1}^{N} \left\| \overrightarrow{\mathbf{v}}_i \right\|},$$

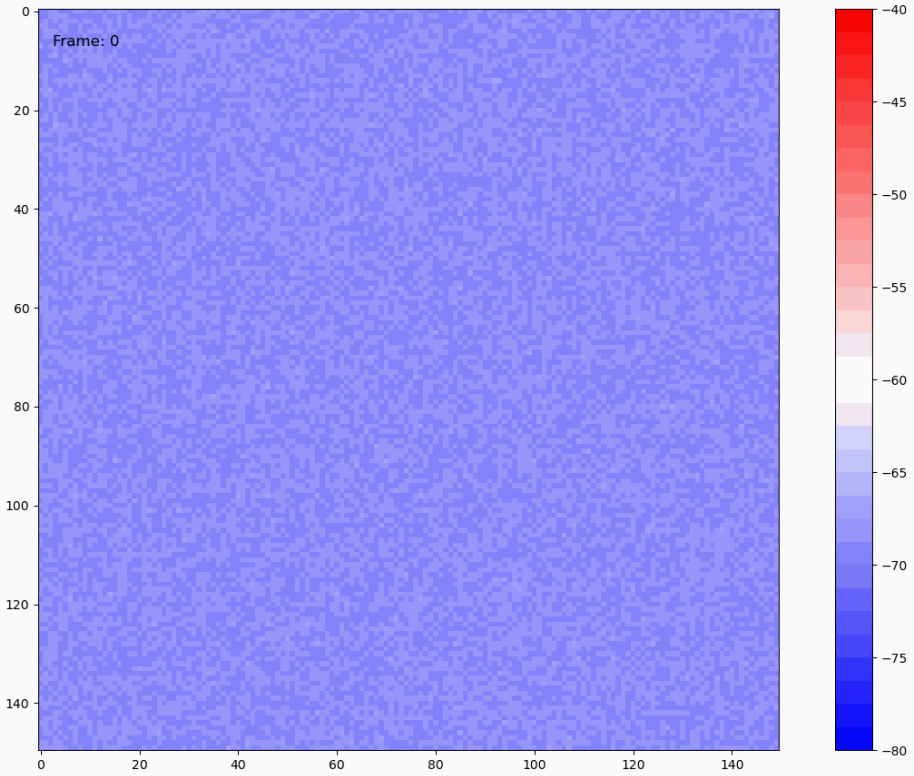
$\mathbf{v}$的大小由前后两帧的同pattern的center决定

为了保证计算采用的前后两帧的center来自于同一个pattern，我们需要对前后两帧的center_list做center的匹配（假设属于同一个pattern的center不会移动太大距离）：

```
def match_patterns(prev_centers: List[Tuple[float, float]],
                   curr_centers: List[Tuple[float, float]],
                   max_distance: float = 20.0) -> List[Tuple[int, int]]:
    """
    匹配两个时间点之间的patterns

    Parameters:
    prev_centers: 前一个时间点的centers
    curr_centers: 当前时间点的centers
    max_distance: 最大匹配距离

    Returns:
    List[Tuple[int, int]]: 匹配的pattern索引对列表 (prev_idx, curr_idx)
    """
    matches = []
    used_curr = set()

    for i, prev_center in enumerate(prev_centers):
                ...
        for j, curr_center in enumerate(curr_centers):
            ...
        if best_match != -1: #这个点在下一帧有匹配的center
            matches.append((i, best_match))
            used_curr.add(best_match)

    return matches
```

以上计算center和匹配center的方式会在两个pattern相距过近时导致本来应该是2个的center合并成一个，并由此影响速度的计算，但是这个问题在模式稳定后并不存在，所以可以不用考虑处理：(下图中绿色十字代表计算出的center位置)

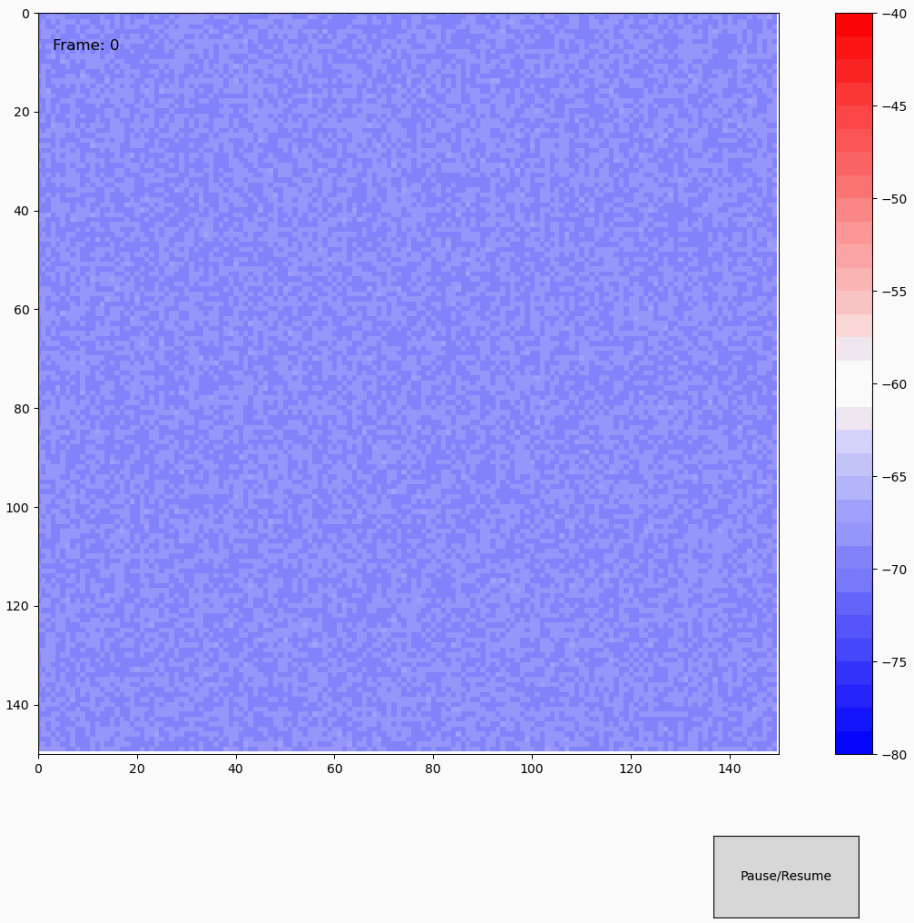

Frame: 0
0
20
40
60
80
100
120
140
−40
−45
−50
−55
−60
−65
−70
−75
−80



匹配了center后就可以直接计算速度:

```python
def calculate_velocities(centers_timeseries: List[List[Tuple[float, float]]],
                         dt,
                         max_distance: float = 10.0) -> List[List[np.ndarray]]:
    """
    计算所有pattern的速度

    Parameters:
    centers_timeseries: 时间序列上的centers
    dt: 时间步长
    max_distance: 最大匹配距离

    Returns:
    List[np.ndarray]: 速度向量列表
    """
    velocities_series = []

    for t in range(int((runtime+100)/dt), len(centers_timeseries) - 1):
        velocities = []
        prev_centers = centers_timeseries[t]
        curr_centers = centers_timeseries[t + 1]

        # 匹配patterns
        matches = match_patterns(prev_centers, curr_centers, max_distance)

        # 计算每对匹配pattern的速度
        for prev_idx, curr_idx in matches:
            prev_pos = np.array(prev_centers[prev_idx])
            curr_pos = np.array(curr_centers[curr_idx])
            velocity = (curr_pos - prev_pos) / dt
            velocities.append(velocity)
        if velocities != []:
          velocities_series.append(velocities)
        else:
          velocities_series.append([0])

        if not matches:
          velocities_series.append([0])

    return velocities_series
```

计算出来的速度结果动画化如下：（红色箭头代表速度向量）



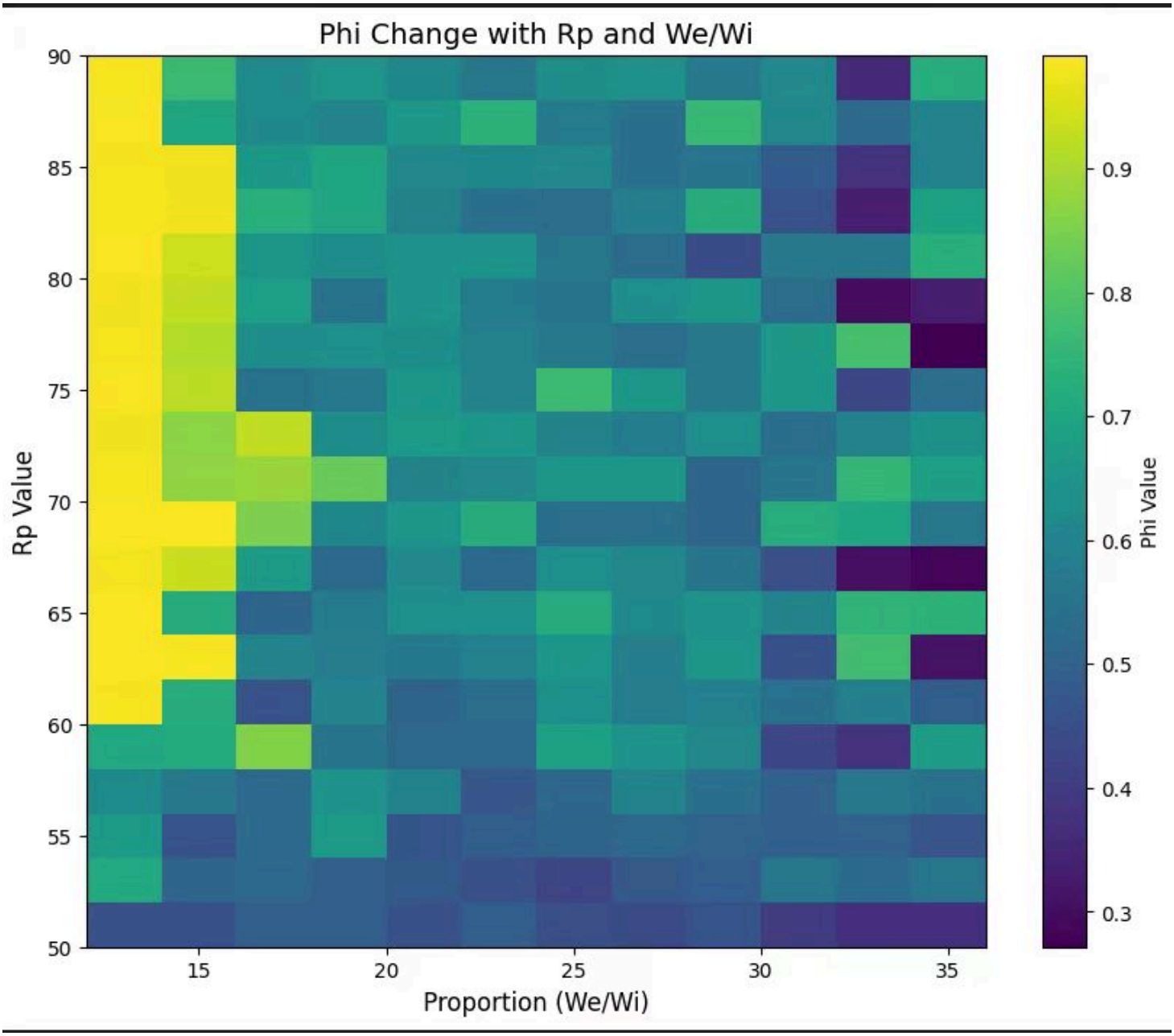
可以看到速度箭头有较大波动，但是在模式稳定后波动较为规律，在ϕ的计算中会抵消。

接下来便是计算ϕ：（由前面观察center和velocities计算情况的观察，实际计算时只取靠后一段时间每秒ϕ的均值，因为此时模型已经发育稳定）

```
def calcu_Phi(data: np.ndarray,
              #dt: float = 1.0,
              max_distance: float = 10.0, type = 0) -> float:
    # 计算时间序列上的所有centers
    centers_timeseries = calculate_centers_timeseries(data)

    # 计算所有velocities
    #print(dt)
    velocities_series = calculate_velocities(centers_timeseries, dt, max_distance)

        ...

    Phi_series = []
    for velocities in velocities_series:
      velocities = np.array(velocities)
      if len(velocities) > 1:
        vector_sum = np.sqrt(np.sum(np.sum(velocities, axis=0)**2))
        sum_magnitude = np.sum(np.sqrt(np.sum(velocities**2, axis=1)))
        Phi = vector_sum / sum_magnitude
      else:
                ...
      Phi_series.append(Phi)

    Phi = sum(Phi_series)/ len(Phi_series)

    return Phi
```

序参量-外部变量的变化参数

对于ϕ关于不应期rp和We/Wi的变化，由如下图所示：



我们可以看到，ϕ变化和rp成正比，和We/Wi成反比。

从动力学系统的角度来分析这个现象，首先，不应期(rp)的影响来看：

1. 当rp较小时，神经元恢复得很快，容易形成高频、无序的放电。这种情况下神经网络中的模式很难形成有序传播，所以序参量φ较小。
2. 随着rp增大，神经元需要更长时间才能恢复，这就限制了它们的放电频率。这种限制反而有助于形成更有序的传播模式，因为一个神经元放电后需要较长时间恢复，就给了兴奋性传播形成稳定波的机会,而不会被快速恢复的神经元产生的无序活动所干扰。这就解释了为什么φ会随rp增加而增大。

另外，从We/Wi比值的影响来看：

1. 当We/Wi较小时，抑制性作用相对较强。这种情况下网络更容易形成局部化的模式,因为强抑制限制了兴奋的扩散。局部化的模式更容易形成有序传播,所以φ较大。

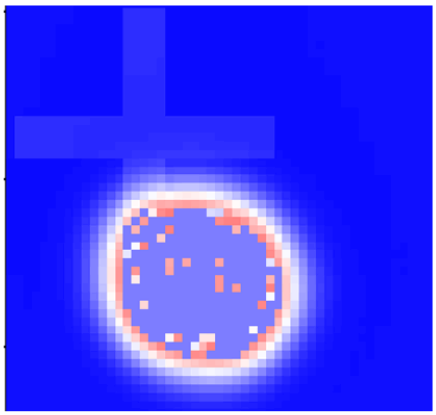
2. 随着We/Wi增大，兴奋性连接增强。此时网络可能会产生更多的新月形波(crescent waves)。这些波的传播方向不一致,导致整体序参量φ降低。

这个现象从神经网络功能的角度也很有意义 - 适当的不应期和抑制性连接强度可以帮助网络形成更有序的信息传播模式,这对于信息加工可能是有益的。但过强的兴奋性连接则可能产生混乱的活动模式,不利于信息处理。

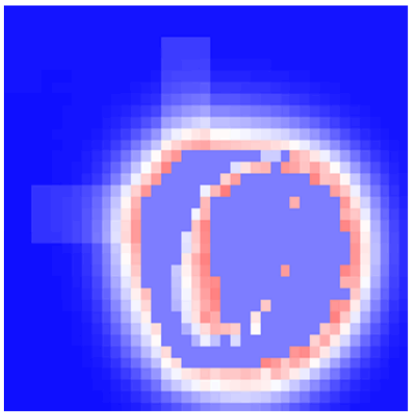
Attention 机制探究

由于提供图片是 numpy 格式的 0,1 二维矩阵（150,150），且与我们之前实验的网络规模相同，我们直接将该 numpy 矩阵作为刺激输入到已经呈现 wave, bump, critical state 的网络上，如图，十字位置外界刺激为 1nA，其余位置外界刺激为 0。这里模拟网络接收具有一定信息的外界刺激下，网络中波包对刺激的编码能力，进而探究人脑的 Attention 机制。
代码如下：

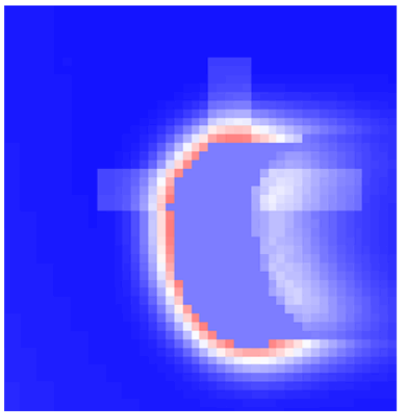
```
image = np.load('image_cross.npy')

if(if_attention):
        image_tensor = torch.tensor(image, dtype=torch.float32).to(device)
        for i in range(int(runtime2/dt)):
            __t__+=dt
            E_potential, E_spike = net.update(
                image_tensor,  # 使用图像刺激输入
                torch.rand((In, In)).to(device) * 0  # 抑制性输入仍然为 0
            )
            voltage_list.append(E_potential.clone().cpu())
```



a.网络呈现自发态时接受刺激

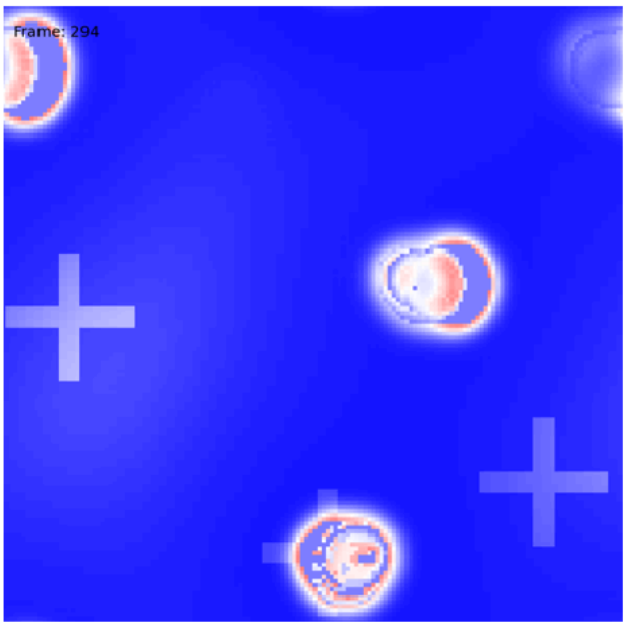


b.网络临界态时接受刺激，波包移动至刺激点

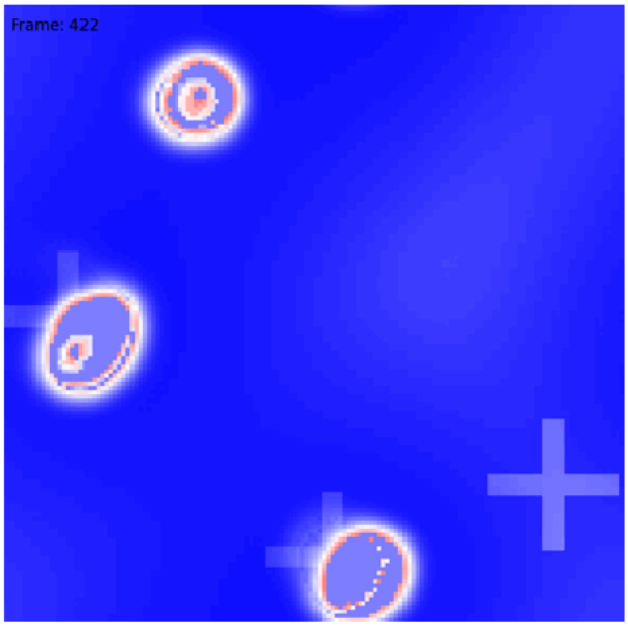


c.网络激发态时接受刺激，行波无法停止在刺激点

当网络呈现bump(rp=5ms)或wave(rp=8ms)，时十字形的刺激对网络原有的波包形态影响不明显，如图a,c。推测原因是bump阶段波包的稳定性较好，spike的频率较快（相对于critical state），所以外界刺激对不敏感。但是也存在特殊情况：在自发态下，如果输入刺激使波包恰好落在刺激的一侧，即使是自发态相对稳定的情况，波包的形状也会出现些微的扰动，向刺激的中心点缓慢移动。
而wave网络行波的运动速度较快，且每个月牙形波包的形态和运动趋势稳定性高，经过刺激点时不会停留，对外界刺激也不敏感。
对于刺激前呈现 critical state 的网络，我们发现对于刺激有比较明显的反应，如图（分别记录了接受刺激瞬间和刺激作用一段时间后的相对稳定状态）：

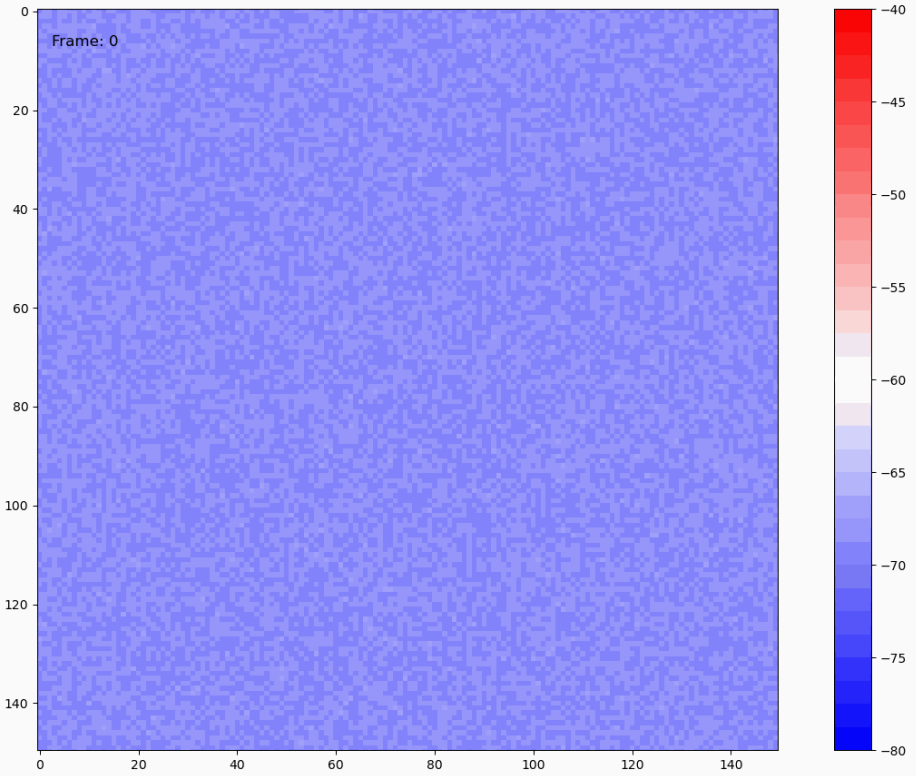


a.接受刺激瞬间bump转换为行波



b.行波运动至刺激点中心时稳定在刺激点

在标准的critical state下（$W_e = 0.12, W_i = 0.06, rp = 6ms$），加入外界刺激，如果先前的bump波包距离刺激点较近，会失去稳态，转换为行波，向前运动，到达刺激点中心时停止大幅度的运动，并围绕刺激点的中心波动，形状的变化明显。当波包有偏离刺激中心点的趋势时，波包spike中心会来自靠近刺激中心的一侧，从而迫使波包向相反方向移动，如图中下部波包。
如果先前波包为行进状态，运动至刺激点后将无法继续向前运动，如图中中部波包。
当波包是临界态且网络受到刺激时波包距离某个刺激中心较近，则该波包会逐渐以椭圆形态向刺激中心靠近，之后维持在刺激点中心附近。



Attention机制的猜测：

由论文的讨论我们可以得知，临界态对于大脑信息处理有特殊意义，它同时表现出局部斑块模式(patchy patterns)和新月形波(crescent waves)，这些模式的传播看似随机，但实际上遵循一定规律分布。它提供了在自发活动和刺激诱发响应之间的最佳平衡，使网络能够快速有效地响应外部输入（如上面动图所示，它在外部的十字刺激下很快得收敛到了十字处），因此允许网络保持足够的可塑性以适应不同的计算需求。

另一方面，它产生的空间相关性模式也与实际观察到的神经活动特征相符。

这个发现的意义在于，我们可以通过统计定义临界态的数值特征，并进一步发现临界态下网络对刺激的相应最快。这给了我们以理解神经网络工作方式的一种启发，也即研究它的动力学形态并关注它的有序/无序状态或者空间活动模式。