Lab 3 Report

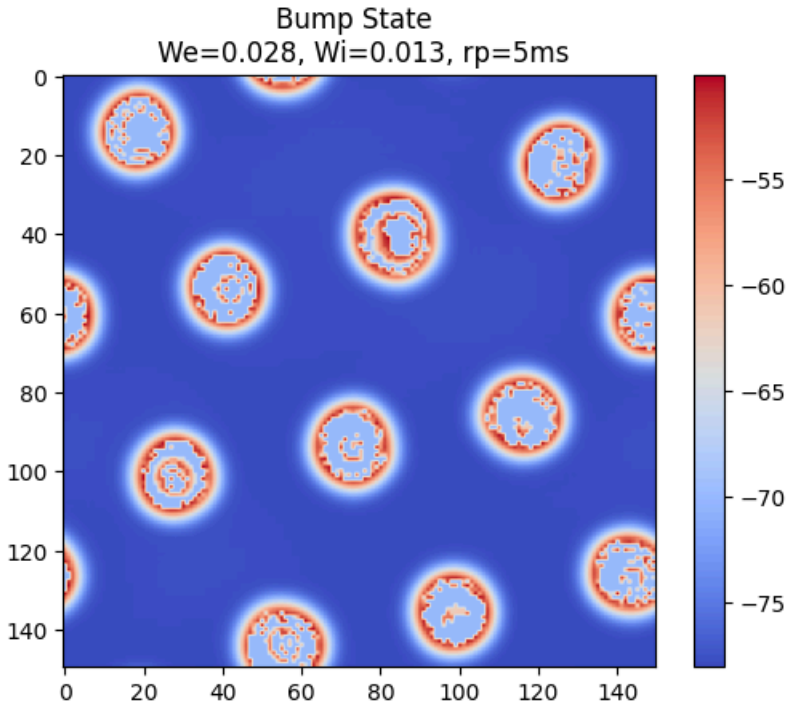
By 徐奕辰 孙佳芮 罗玥萦

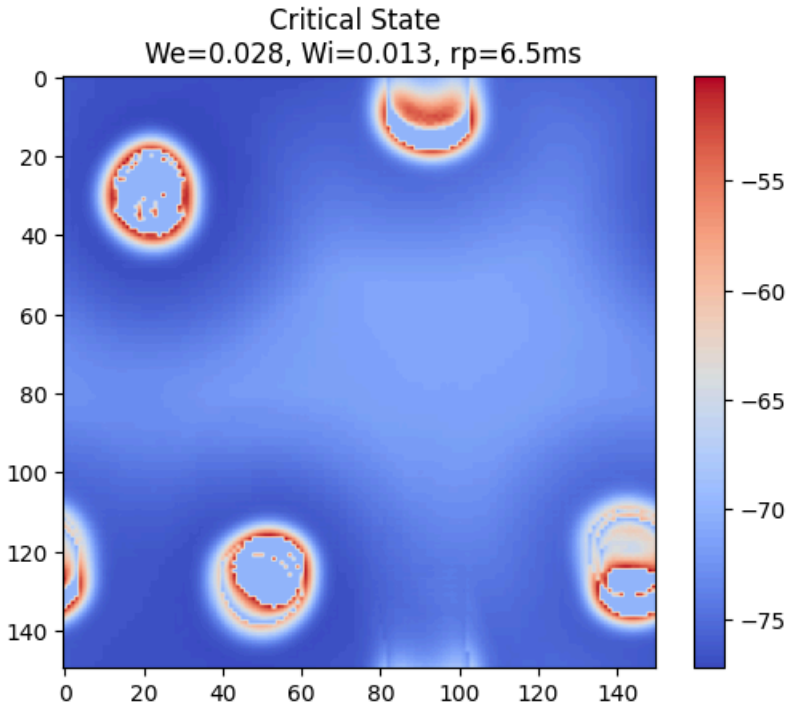
问题1 CMSA网络的搭建

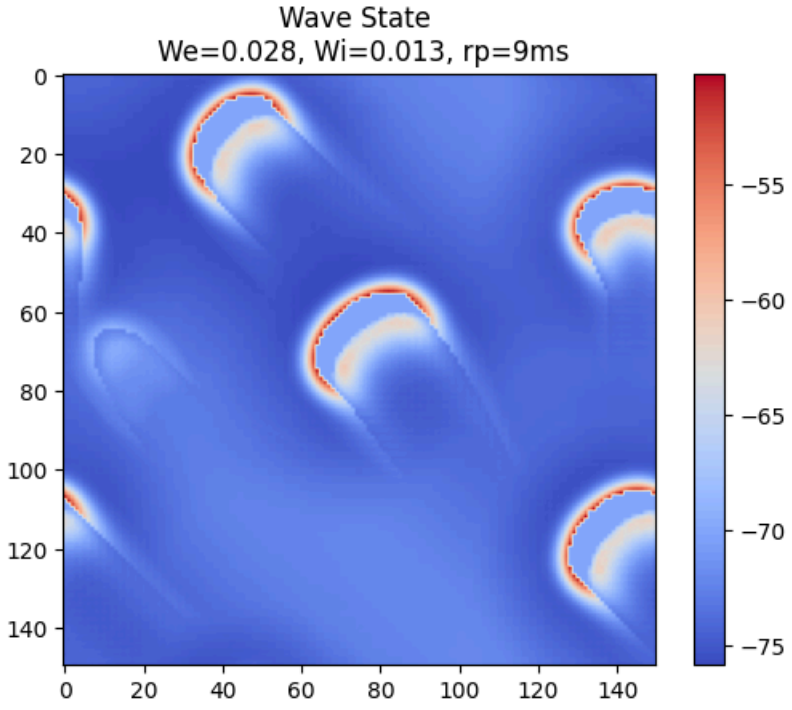
在本问题中，我们基于LIF点神经元模型搭建神经元层，基于AMPA和GABA两类突触进行指数衰减型突触建模。我们将MSynapse调整为75，sigma调整为200。在调试找到合适的参数后，我们通过改变不应期长度，发现

- 在不应期较小时（如rp=5），网络状态为bump；
- 而当不应期较大时（如rp=9），网络状态为wave；
- 当rp=6.5或7时，网络状态可能为critical（也可能转化为为bump或wave，具体取决于开始外界给予的随机输入），体现了在临界状态下网络对外界输入敏感度高的状态。不应期对系统状态（序参量）的影响分析见问题2.

具体而言，序参量以及pattern的形态反映了系统的对称性。因此更有利于维持系统的对称性：假设一个兴奋性神经元被激活，下一个时间步它会对称地激活它周围的神经元（包括兴奋性和抑制性），进入不应期之后该激活效果消失，如此与其他神经元发生耦合。而当模拟开始时给予系统的随机输入导致了神经发放模式的产生，当出现一段联通的兴奋性神经元时，它就很容易以”线“的形式激活相邻的神经元，进而宏观上产生pattern的前进（呈半月形）。而当模型的参数设置倾向于保持对称性时，网络则倾向于回复到对称性较高的状态，使得系统总体的能量较低，表现为bump。




Critical State
We=0.028, Wi=0.013, rp=6.5ms
0
20
40
60
80
100
120
140
−55
−60
−65
−70
−75


Wave State
We=0.028, Wi=0.013, rp=9ms
0
20
40
60
80
100
120
140
−55
−60
−65
−70
−75

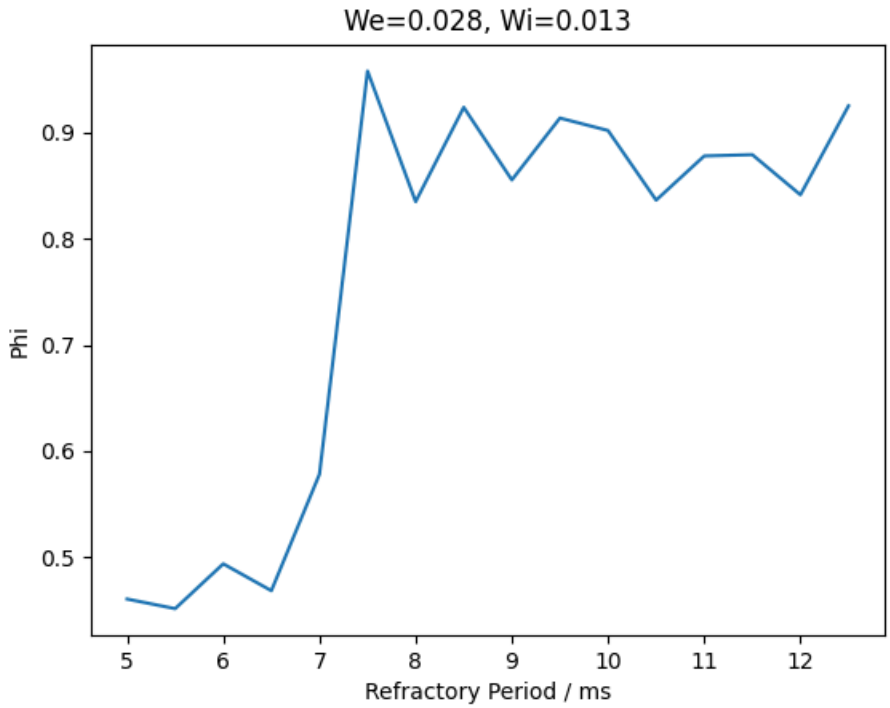
- Code

```
class LIFlayer:
    def update(self, input:torch.Tensor):
        assert input.shape == self.shape
        #TODO 请你完成膜电位、神经元发放和不应期的更新
        # 计算电压变化率 dV/dt
        dV_dt = (-self.gL * (self.potential - self.reset_value) + input) / self.membrane_capacitance
        self.potential += dV_dt * dt
        is_refractory = (__t__ - self.spike_time) < self.refractory  # 在不应期内
        is_threshold_crossed = (self.potential >= self.threshold) & ~is_refractory  # 满足发放阈值且不在不应期内
        self.spike = is_threshold_crossed.float()
        self.spike_time[is_threshold_crossed] = __t__
        self.potential[is_threshold_crossed] = self.reset_value
        self.potential[is_refractory] = self.reset_value

        return self.potential, self.spike

class Synapseslayer:
    def gaussian(self, n, W, sigma):
        #TODO 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）
        center = n // 2
        X, Y = torch.meshgrid(torch.arange(n)-center, torch.arange(n) - center)
        gaussian = W * torch.exp(-(X**2 + Y**2)/(sigma))
        gaussian = gaussian.to(device)

        return gaussian
```

问题2 相变过程的探究

Part 1&2 序参量的影响因素

在本问中，相比于作业参考的原文献中对We对相变过程的影响探究，我们选取不应期长度(rp)，探究它对系统的序参量的影响。我们将不同seed得到的结果进行平均得到了最终的图像。



对于原文献中的参数设置，在其他参数固定的情况下，We较小时系统处于bump state，较大时系统处于wave state。这可能是因为当We较大时，兴奋性神经元输出的刺激电流波动更大，因此更容易造成系统对称性破缺。

对于不应期，我们通过遍历改变不应期长度进行探究。特别的，由于系统的模拟步长为0.5，我们对不应期的步长也选取为0.5。我们发现

- 在不应期较小时（如rp=5），网络状态为bump；
- 而当不应期较大时（如rp=9），网络状态为wave；
- 当rp=6.5或7时，网络状态可能为critical（也可能转化为为bump或wave，具体取决于开始外界给予的随机输入），具体而言，当不应期较小时，单个神经元在进入不应期后经过较短的时间就能够重新被激活，因为单个神经元在短时间内就能被重新激活，因此系统更不容易发生对称性破缺。而当不应期较大时则反之。

特别的，在特定的初始输入条件下，wave state的网络序参量较小，观察后我们发现这常常对应着网络中存在相反方向运动的pattern，它们的形成过程是：在外界输入停止，网络由不稳定的状态逐渐演化到稳定状态的过程中，有许多wave按照任意由初始条件决定的方向移动，在这个过程中和其他wave在延申的未来轨迹上产生交集，往往会有两种后果：

1. 其中一个wave的兴奋性被另一个wave导致的周围神经元的抑制性所压制，因此这个wave消失；
2. 两个wave势均力敌，那么它们的运动方向就会被彼此影响，最终处于二者运动方向相平行的状态。

- Code

```
def flood_fill(image, x, y, threshold, visited):
    stack = [(x, y)]
    region = []
    contains_above_threshold = False
    width = image.shape[0]

    while stack:
        px, py = stack.pop()
        if (px, py) not in visited and image[px, py] > threshold:
            visited.add((px, py))
            region.append((px, py))
            if image[px, py] > -55:
                contains_above_threshold = True
            for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1), (-1, -1), (-1, 1), (1, -1), (1, 1)]:
                nx, ny = (px + dx) % width, (py + dy) % width
                if (nx, ny) not in visited:
                    stack.append((nx, ny))
    if not contains_above_threshold:
        return []

    return region

def calcu_center(image):
    visited = set()
    centers = []
    threshold = -74
    width = image.shape[0]
    for x in range(image.shape[0]):
        for y in range(image.shape[1]):
            if (x, y) not in visited and image[x, y] > threshold:
                region = flood_fill(image, x, y, threshold, visited)
                if region:
                    region_array = np.array(region, dtype=np.float32)
                    x_coords = region_array[:, 0]
                    y_coords = region_array[:, 1]
                    if np.max(x_coords) - np.min(x_coords) > width / 2:
                        x_coords[x_coords < width / 2] += width
                    if np.max(y_coords) - np.min(y_coords) > width / 2:
                        y_coords[y_coords < width / 2] += width

                    x_center = np.mean(x_coords) % width
                    y_center = np.mean(y_coords) % width
                    centers.append([x_center, y_center])

    return centers

def wrap_distance(p1, p2, width):
    dx = min(abs(p1[0] - p2[0]), width - abs(p1[0] - p2[0]))
    dy = min(abs(p1[1] - p2[1]), width - abs(p1[1] - p2[1]))

    return np.sqrt(dx**2 + dy**2)

def compute_velocity(last, current, width):
    dx = current[0] - last[0]
```

```
    dy = current[1] - last[1]
    if abs(dx) > width / 2:
        dx = dx - np.sign(dx) * width
    if abs(dy) > width / 2:
        dy = dy - np.sign(dy) * width

    return (dx, dy)

def pair_centers(last_centers, current_centers, width, max_distance=5, margin=2):
    pairs = []
    used_current = set()
    for last in last_centers:
        distances = []
        for i, current in enumerate(current_centers):
            if i not in used_current:
                distance = wrap_distance(last, current, width)
                distances.append((distance, i))
        if not distances:
            continue

        distances.sort()
        best_distance, best_match = distances[0]
        second_best_distance = distances[1][0] if len(distances) > 1 else float('inf')

        if best_distance <= max_distance and (second_best_distance - best_distance) >= margin:
            pairs.append((last, current_centers[best_match]))
            used_current.add(best_match)

    return pairs


def calcu_Phi(data: np.ndarray, max_distance=5, margin=2):
    width = data.shape[1]
    last_center_list = calcu_center(data[0])
    phi_list = []

    for t in range(1, data.shape[0]):
        current_center_list = calcu_center(data[t])
        pairs = pair_centers(last_center_list, current_center_list, width, max_distance=max_distance, margin=ma
        velocity_list = []

        for last, current in pairs:
            velocity = compute_velocity(last, current, width)
            velocity_list.append(velocity)

        if len(velocity_list) > 0:
            velocities = np.array(velocity_list, dtype=np.float32)
            Phi = np.linalg.norm(velocities.sum(axis=0)) / np.sum(np.linalg.norm(velocities, axis=1))
            phi_list.append(Phi)

        last_center_list = current_center_list

    if len(phi_list) == 0:
        return 0
```

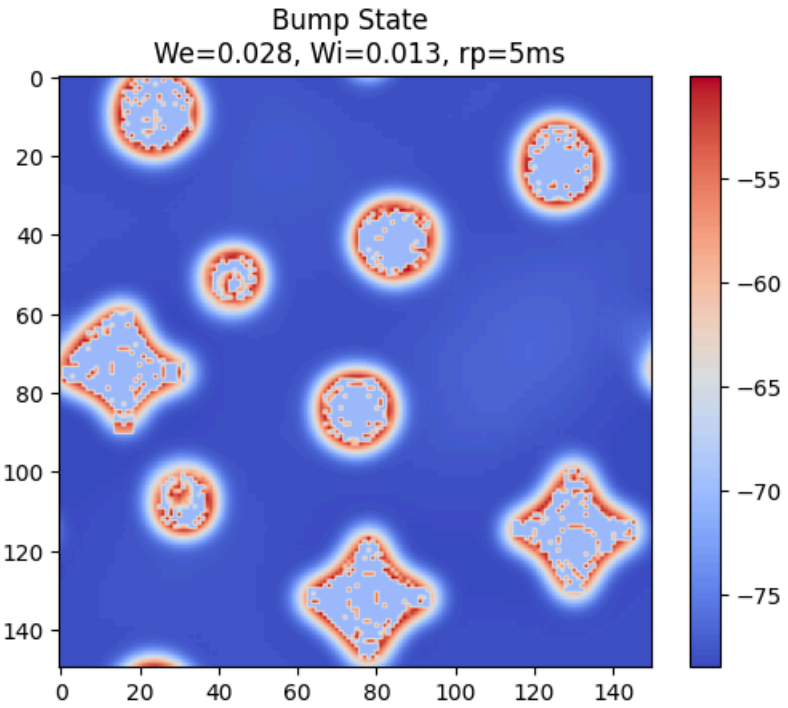
```
    average_phi = np.mean(phi_list)
    return average_phi

### 进行模拟，绘制图像 ###

from tqdm import tqdm
rp_range = np.arange(5, 13, 0.5)
torch.manual_seed(0)
np.random.seed(0)

for seed in range(0, 101, 10):
    print("seed =", seed)
    torch.manual_seed(seed)
    np.random.seed(seed)
    Phi_list = []
    inputE = torch.randn(int(runtime1/dt),En,En).to(device)*5
    inputI = torch.randn(int(runtime1/dt),In,In).to(device)*5

    __t__ = 0

    for rp in tqdm(rp_range):
        net = Network(En,In,rp=rp,We=We,Wi=Wi,Msynapse=msynapse)
        voltage_list = []

        for i in range(int(runtime1/dt)):
            __t__+=dt
            E_potential, E_spike= net.update(inputE[i], inputI[i])

        for i in range(int(runtime/dt)):
            __t__+=dt
            E_potential, E_spike= net.update(torch.rand((En,En)).to(device)*0, torch.rand((In,In)).to(device)*0
            voltage_list.append(E_potential.clone().cpu())

        voltage_list = voltage_list[-100:]
        Phi = calcu_Phi(np.array([v.numpy() for v in voltage_list]))
        Phi_list.append(Phi)

    np.save(f'Seed={seed}',Phi_list)
    plt.figure(seed)
    plt.plot(rp_range, Phi_list)
    plt.xlabel('Refractory Period / ms')
    plt.ylabel('Phi')
    plt.title(f'We={We}, Wi={Wi}')
    plt.savefig(f'Rp-We={We}-Wi={Wi}-Seed={seed}.png')
```

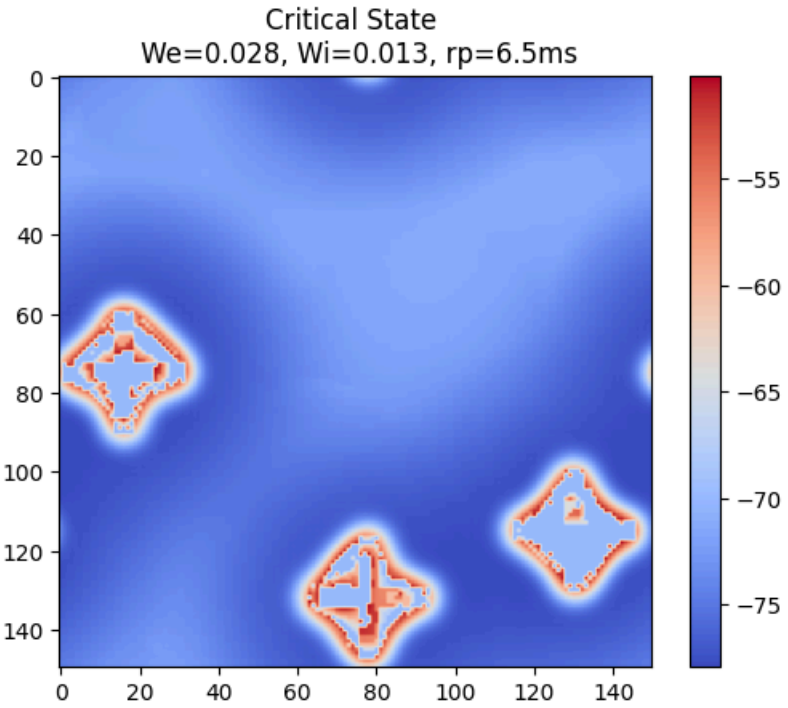
Part 3 临界态对注意力模式的影响

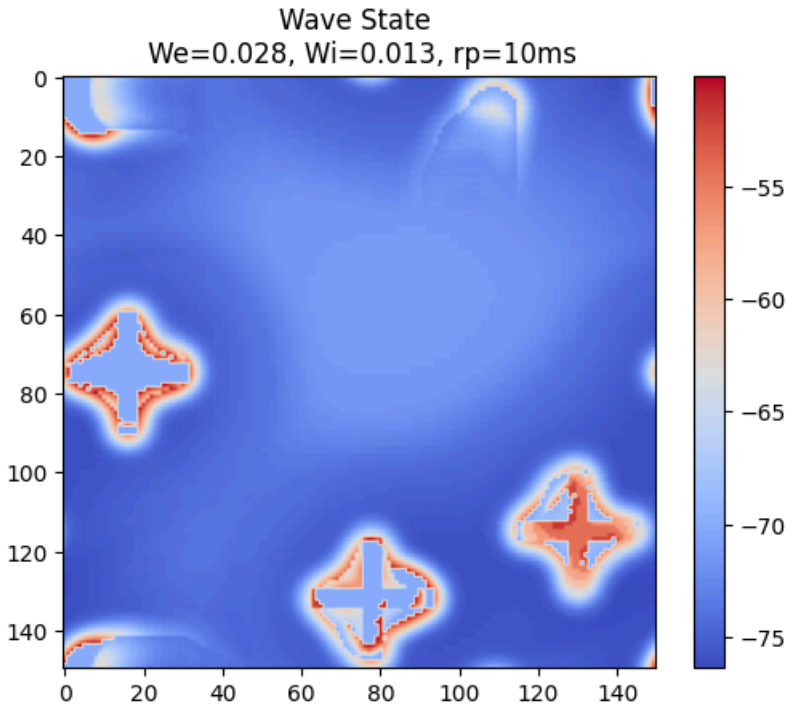
在本部分，我们在系统处于稳定state后施加外界图像刺激，观察到以下现象：

- pattern不会在刺激输入后消失，而是通过刺激的引导重新分布到高刺激强度区域（在本题的刺激下，pattern看起来被”吸“到了外界刺激图像上。特别的。由于使用的刺激十字的大小和pattern的大小相似，所以pattern在被吸引到十字上后会呈十字形。

- 具体而言，pattern首先被外部刺激拉动，逐渐靠近刺激图案中的兴趣区域；聚焦过程中，模式的形状和传播方向都发生调整，最终“捕获”在刺激的兴趣区域内。这一动态表现为从模式的“移动”到“聚焦”的过程，揭示了模式被重新分布的时间序列特性。







特别的，通过比较在bump state, critical state和wave state三种状态下的注意力模式，我们发现：

- 对于bump state，外界图像刺激无法直接激活原来未被激活的部分，只能通过圆形的bump pattern的边缘与刺激覆盖区域的神经元的互作，把有接触的pattern”吸“过来，用时较长。如果图像刺激周围原先没有bump pattern则无法产生对应的”注意力“，活动模式局限于固定的局部区域，远离目标区域的模式很难被重新分布。这反映出活动过于局部化，缺乏灵活的传播特性。
- 对于critical state，外界图像刺激可以直接激活原来未被激活的部分，而原先附近的pattern如果不在被刺激处，则会在网络的神经元的相互作用下逐渐消失，较远的pattern则受影响较小。这反映出临界态作为系统对称性状态两端之间的过渡点，使系统在此状态下可以处理最广泛范围的刺激强度。此外，临界态的活动模式对外部输入高度敏感，能够快速捕捉刺激信号，并相应调整活动的传播路径；模式在受到刺激时，能以较低的“调节成本”快速转变为与刺激相匹配的形状和位置。
- 对于wave state，外界刺激处会立刻产生斑块，但相比critical state图像刺激处持续发放，wave state中图像刺激处的神经元发放会呈现频闪状态。

为什么大脑应该处于临界态?

在临界态下，系统具有以下性质：

1. 临界态是有序状态和无序状态的过渡点，系统在此状态下具有最大的动态范围，即能够有效响应最广泛范围的刺激强度。这种特性使大脑在应对复杂、变化多端的环境刺激时表现出高度适应性。
2. 临界态下，神经网络对外部输入的敏感性最高，能够快速捕获并处理外部刺激。模式的传播和调节在此状态下最为灵活，刺激能够更快、更有效地引导活动模式聚焦到目标区域。
3. 临界态允许多个活动模式在网络中独立传播，并能够被刺激同时调节。这种 分布式并行处理 特性极大提高了大脑的计算效率。
4. 临界态下的自发活动具有复杂的空间和时间结构，呈现出 波状传播、斑块活动和共激活现象。这种结构为外部刺激的调节提供了丰富的基础。

- Code

```
image = np.load('image_cross.npy')
En = 150 # 兴奋性神经元网格边长数
In = 75 # 抑制性神经元网格边长数，En与In之比需要为整数
runtime1 = 50  # 有外界输入的时间
runtime2 = 300  # 自发动力学的时间
runtime3 = 100  # 有外界输入的时间
rp = [5, 6.5, 10]  # 不应期时间
We =  0.028 # 兴奋性连接权重
Wi =  0.013 # 抑制性连接权重
caption = ["Bump State", "Critical State", "Wave State"]

for rt in range(3):
    __t__ = 0
    net = Network(En,In,rp=rp[rt],We=We,Wi=Wi, Msynapse=75)
    voltage_list = []
    torch.manual_seed(0)
    np.random.seed(0)
    image = torch.tensor(image, dtype=torch.float32)
    input_amp = 5
    image_inputE = image * input_amp

    for i in range(int(runtime1/dt)):
        __t__+=dt
        E_potential, E_spike= net.update(torch.rand((En,En)).to(device)*5, torch.rand((In,In)).to(device)*5)  #
        voltage_list.append(E_potential.clone().cpu())

    for i in range(int(runtime2/dt)):
        __t__+=dt
        E_potential, E_spike= net.update(torch.rand((En,En)).to(device)*0, torch.rand((In,In)).to(device)*0)
        voltage_list.append(E_potential.clone().cpu())

    for i in range(int(runtime3/dt)):
        __t__+=dt
        E_potential, E_spike= net.update(image_inputE.to(device), torch.rand((In,In)).to(device)*0)  # 平均外界输
        voltage_list.append(E_potential.clone().cpu())

    plt.imshow(E_potential.cpu(), cmap='coolwarm')
    plt.title(f'{caption[rt]} \n We={We}, Wi={Wi}, rp={rp[rt]}ms')
    plt.colorbar()
    plt.show()
```