4G3 Computational Neuroscience

Part IIB - 4G3 Coursework 2: Recurrent neural network dynamics

Edited By

Hidden Person
5656A

University Of Cambridge

Introduction

The brain is an extremely complex dynamical system. To help understand it computational neuroscience uses network models to analyze behaviour and construct networks that can perform tasks. These models can vary from extremely complex models like Blue Gene or they can be simplified for example by using rates instead of spikes.

Here we will be looking at a simplified recurrent network model of the primary visual cortex (V1). The intention is to investigate how it can represent a specific feature - in this case the orientation of an input and how recurrent connectivity affects the reconstruction error.

The model firing rate is governed by the following equation:

$$\tau \frac{d\mathbf{r}}{dt} = -\mathbf{r} + W\mathbf{r} + B\mathbf{h}(\theta)\delta(t) \tag{1}$$

Where τ is the characteristic neuronal time constant, $\mathbf{r}(t)$ represents the firing rate of all neurons in the network at time t, W is the recurrent connectivity matrix, B is the input matrix, $\mathbf{h}(\theta)$ is the input feature vector and $\delta(t)$ is the Dirac delta function. When $t < 0$, $r(t) = 0$.

The input feature vector $\mathbf{h}(\theta)$ is a vector of length m that represents the input orientation with the i^{th} element given by:

$$h_i(\theta) = \mathbb{V}(\phi_i - \theta) \quad \text{where } \mathbb{V}(z) = \exp\left(\frac{\cos(z) - 1}{\kappa^2}\right) \tag{2}$$

Where ϕ_i is the preferred orientation of neuron i and κ is a parameter that controls the width of the Gaussian ($\mathbb{V}$ iz like an unnormalised Gaussian).

A noisy readout is then given by:

$$\bar{\mathbf{o}}(t) = Cr(t) + \sigma\epsilon(t) \tag{3}$$

Where $\bar{\mathbf{o}}(t)$ is the noisy readout, C is the readout matrix, σ is the noise level and $\epsilon(t)$ is a Gaussian white noise process. This noisy readout is decoded to produce an estimate $\hat{\theta}$ by: $\hat{\theta} = atan\left(\frac{\sum_i o_i(t)\sin\phi_i}{\sum_i o_i(t)\cos\phi_i}\right)$ and the reconstruction error is given by $acos(\cos(\hat{\theta} - \theta))$.

Four different recurrent connectivity matrices are used in this report: **Zero**: All elements are zero. **Random Symmetric**: $W = \mathbb{R}(\bar{W} + \bar{W}^T, \alpha)$ where $\bar{W}$ is a random matrix with elements normally distributed with mean 0 and standard deviation 1. **Symmetric Ring**: $\bar{W}_{ij} = \mathbb{V}(\phi_i - \phi_j)$ (the ring model defined in the lectures) and $W = \mathbb{R}(\bar{W}, \alpha)$. **Balanced Ring**: $\begin{bmatrix} \bar{W}, -\bar{W} \\ \bar{W}, -\bar{W} \end{bmatrix}$ where $\bar{W} = \mathbb{R}(\text{Symmetric Ring W}, \alpha)$ (the E and I ring model defined in the lectures with $\gamma = 1$) where $\mathbb{R}(W, \alpha)$ makes the largest eigenvalue of W equal to α.

Questions

For all questions use the following parameters: $\tau = 20 * 10^{-3}s$, $m = 200$, $n = 200$, $B = I$, $C = I$, $\sigma = 1$, $\kappa = \pi/4$, $\alpha = 0.9$ except for the balanced ring where $n = 400$, $B = [I_m 0_m]^T$, $C = [I_m 0_m]$.

Q1 Investigating Rates

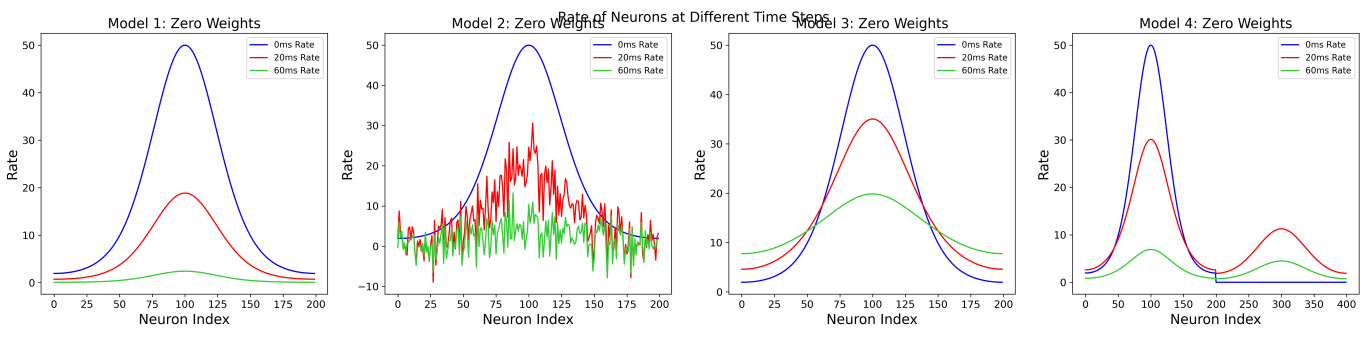

Figure 1: The rate of neurons in the network at $t = [0^+, 20ms, 60ms]$ to a $\theta = \pi$ input.

Fig 1 shows the rate of neurons in the network at $t = [0^+, 20ms, 60ms]$ to a $\theta = \pi$ input. As we can see at 0+ all rates are the same for neurons 0-200. This is because $\mathbf{r}(0) = 0$ and so $\mathbf{r}(0^+) = B\mathbf{h}(\pi)$ where B is the same for all networks for neurons 0-200. We can also see that all rates decay as the time increases. This is because all networks max real eigenvalues of $W - I$ is less than 0 so the recurrent activity is not enough to overcome the decay. Lastly all rates have a bump at neuron 100 (and neuron 300 for the balanced ring). This is because the input feature vector $\mathbf{h}(\pi)$ has a peak at neuron 100 (and neuron 300 for the balanced ring) as this is the neuron that is aligned with the input while all other neurons get more mis-aligned as the difference in neuron index increases so $cos(z_i)$ decreases and so $h_i(\pi)$ decreases.

If we look at the different networks we can see the zero recurrence network has the fastest decaying rate. This is because there is no recurrent connectivity and so the only thing that is affecting the rate is the decay rate. The random symmetric network is extremely noisy. While the symmetric ring network seems to have the slowest decay rate.

The symmetric ring network is the most different. It has inhibition and excitation neurons and acts as non-normal matrices. These inhibition neurons (neurons 200-400) do not have any connections to the input feature, so have $r(0^+) = 0$. They then get excited by the recurrent activity of the excitatory neurons (neurons 0-200) which are not 0. However this causes the rate of the excitatory neurons to decay faster so the decay rate of model 4 is faster than the decay rate of model 3 (very evident at 60ms).

Q2 Noise of Random Symmetric Network

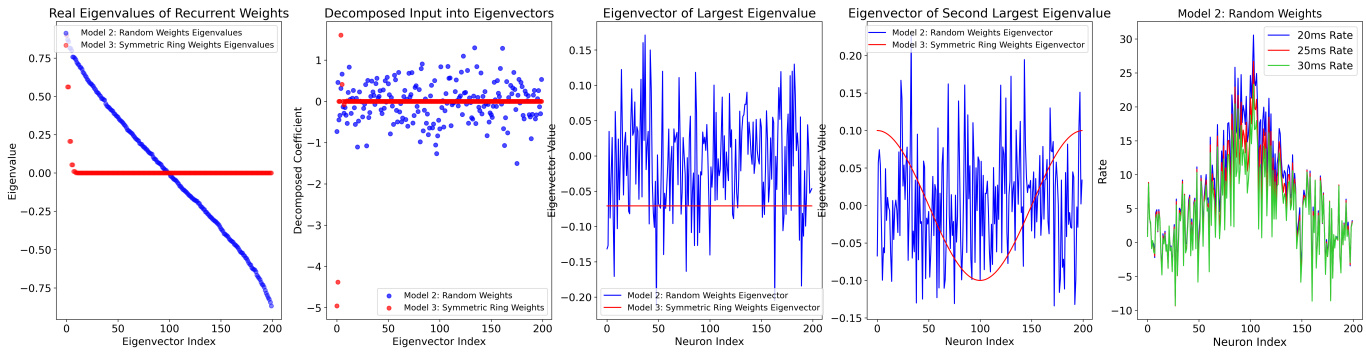

Figure 2: a) Sorted eigenvalues of random symmetric and symmetric ring networks. b) Decomposed coeffecients of $\mathbf{h}(\pi)$ for the 2 networks. c) Eigenvector of lasrgest eigenvalue for both networks. d) Eigenvector of second largest eigenvalue for both networks. e) Rate of neurons in the random symmetric network at $t = [20ms, 25ms, 30ms]$ to a $\theta = \pi$ input.

When looking at the rates of both networks we can see that the random symmetric network has a lot of noise. To understand this we need to look at the eigenvalues of the recurrent matrix W and compare them to the eigenvalues of the symmetric ring network.

Specifically we can rewrite the rate $r(t)$ as:

$$\mathbf{r}(t) = \sum_{i=1}^{n} a_i(t)\mathbf{v_i} \tag{4}$$

where $a_i(t)$ is a coefficient and v_i is an eigenvector of W (as eigenvectors of W and $W - I$ are the same). Plugging this into the rate equation and ignoring initial input we get:

$$\tau \sum_{i=1}^{n} \dot{a}_i(t)\mathbf{v_i} = (W - I)\sum_{i=1}^{n} a_i(t)\mathbf{v_i} = \sum_{i=1}^{n} a_i(t)(\lambda_i - 1)\mathbf{v_i} \tag{5}$$

$$a_i(t) = a_i(0)\exp\left(\frac{(\lambda_i - 1)t}{\tau}\right), \ \mathbf{r}(t) = \sum_{i=1}^{n} a_i(0)\exp\left(\frac{(\lambda_i - 1)t}{\tau}\right)\mathbf{v_i} \tag{6}$$

Where λ_i is the i^{th} eigenvalue of W and v_i is the corresponding eigenvector. So the decay rate of the network is determined by the eigenvalues of W.

The eigenvalues of the random symmetric networks are shown in Fig 2a. Only real parts are shown as for the random symmetric network all eigenvalues are real (property of symmetric matrices) and for the symmetric ring network all eigenvalues have very small imaginary parts (1e-18, caused by numerical error). As we can see the largest eigenvalue is 0.9 for both networks (as we set it to this). However the eigenvalues of the random ring network quickly decay to 0. The eigenvalues of the random symmetric however have a very large spread.

Now we can look at the coefficients $a_i(0)$. These will satisfy the following equation:

$$\sum_{i=1}^{n} a_i(0)\mathbf{v_i} = B\mathbf{h}(\pi) \tag{7}$$

where in these cases $B = I$. Fig 2b shows the coefficients of the eigenvectors for both networks. In the random symmetric network the coefficients are randomly distributed and have a large spread over all eigenvectors. However the symmetric ring network has 2 coefficients that dominate (and one more that is slightly larger than the others). These coefficients correspond to the eigenvectors of the largest and second largest eigenvalues.

In fig 2 c and d we visualise the the eigenvectors corresponding to the largest and second largest eigenvalues. In the random symmetric network they are random and have no structure. However in the symmetric ring they are flat or have a shape that is similar to the input feature vector $\mathbf{h}(\pi)$. This is why the rate appears noisy for the random network and not the ring network. As time progresses, different coefficients decay at different rates. The eigenvectors with large initial coefficients and small decay rates (this is more important) dominate the rate and so the rate will look like them. So in the random symmetric network the rate will look like a random eigenvector while in the ring it will appear like the input.

Fig 2e shows the rate of the random symmetric network at $t = [20ms, 25ms, 30ms]$. It acts as a sanity check. Specifically the rate at each timestep looks like random fluctuations. However our analysis so far says that since the eigenvalues have no imaginary part there should be no frequencies and if eigenvectors are causing the random noise the shape of the rate should nt change. We see this in rate. Between timestep only the amplitude of the rate changes, the peaks and trough of the rate are still in the same place. Notably this is different to the random symmetric network that is mentioned in the lectures. That is because in there while the recurrent connectivity is basically the same, the rate was getting affected by random noise at each timestep and getting amplified, however here only the readout is getting affected by noise.

Q3 Amplitude of Model Response

From fig 2b we can also see why the amplitude of the symmetric ring network is larger than the other networks. For the zero network the amplitude is the smallest as the only thing that is affecting the rate is the decay rate and there is no recurrent connectivity. For the other networks to have a high amplitude at 60ms we want a high initial coefficient for eigenvectors with a small decay rate (large positive eigenvalue). The symmetric ring network has a much larger coefficient for the largest eigenvalue (which is 0.9 for all networks) than the random symmetric network (about 5 times larger) while having a similar eigenvector norm so it has a larger amplitude (less than 5 times due to other coefficients and slightly smaller eigenvector norm). The balanced ring network has extremely small eigenvalues (less than 1e-10 real part) and so solving for the coefficients gives us a very large coefficient (even trying to re-condition by adding a small identity matrix to W does not help). However as the eigenvalues are so small the decay rate will be faster (approximately $-1/\tau$ instead of $-0.1/\tau$) and so the amplitude will be smaller than the symmetric ring network.

There is also one simpler explanation for why the symmetric ring network has a larger amplitude than all other networks. It is the only network that is non-zero and has no negative values. So rates can only excite each other and not inhibit each other.

Q4 Decoding Error

Fig 3a shows the decoding error averaged over 100 trials. Due to the way we are decoding, we want the rate to be as close as possible to the shape of the input feature vector $\mathbf{h}(\pi)$ and we want the rate amplitude to be as large as possible so readout noise does not affect the decoding.

Across all networks the decoding error starts of at 0 - as at $t = 0^+$, $C\mathbf{r}(0^+) = B\mathbf{h}(\pi)$. However as time progresses the decoding error increases. This is because the rate decays and so the readout becomes less accurate. The zero network decays the fastest and so we would expect it to have the largest decoding error. However actually the

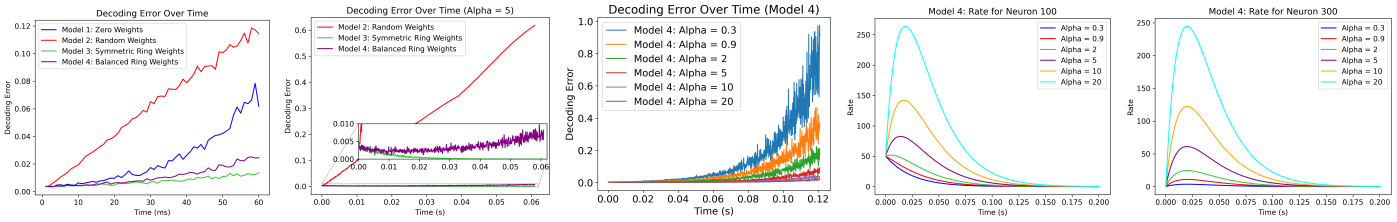

Figure 3: a) Decoding error averaged over 100 trials, $\alpha = 0.9$. b) Decoding error averaged over 100 trials, $\alpha = 5$. c) Decoding Error for model 4 for different α. d) Rate of neuron 100 for different α. e) Rate of neuron 300 for different α.

random symmetric network has the largest decoding error. This is because the shape of the rate is not similar to the input feature vector $\mathbf{h}(\pi)$ and so the decoding is not accurate.

Next the decoding performance of the symmetric ring network is better than the balanced ring network as it decays slightly slower.

Q5 $\alpha = 5$ for balanced ring network

Increasing α to 5 causes a couple of things to happen. Firstly the absolute value of the decoding error is much lower than all other models which is good. Secondly the decoding error temporarily decreases for the first 10ms and lastly the long term decoding error still starts increasing after 10ms.

The balanced ring network is a special case where the recurrent matrix is non-normal matrix. This is a kind of matrix where $WW^T \neq W^TW$. This kind of matrix can cause transient amplification when the excitatory and inhibitory have imbalances. However, here there is no imbalance yet there is still a small decreases in decoding error for the initial 10ms which could only be caused by rate temporarily increasing.

The reason for this is due to the initial conditions of the network. When $t = 0^+$ only the excitatory neurons are firing and the inhibitory neurons are not firing. These get multiplied by the recurrent connectivity matrix $\bar{W}$ which has $\alpha = 5$. As $\alpha > 1$ this cause the rate of the excitatory neurons to increase and this leads to the decoding error decreasing. However as the rate of the excitatory neurons increases the rate of the inhibitory neurons also increases (as they are connected to the excitatory neurons) until they are equal. And eventually since the eigenvalues of the overall matrix are all less than 1, the rate will eventually start decaying.

This is confirmed by looking at the rates of neuron 100 and 300 in fig 3c and d. Neuron 100 is the excitatory neuron and neuron 300 is the inhibitory neuron and these are the neurons with the largest firing rate for this input. As we can see initially the excitatory neuron has a higher firing rate than the inhibitory neuron. So they both increase until they are equal and then they decrease.

Q6 $\alpha = 5$ for other networks

Fig 3b shows the decoding error for the other networks with $\alpha = 5$. One thing to note is when $\alpha < 1$, $\lim_{t\to\infty} r(t) \to 0$ however when $\alpha > 1$, $\lim_{t\to\infty} r(t) \to a_0(0)e^{t(\lambda_0-1)/\tau}v_0 \to \infty$ where λ_0 is the largest eigenvalue of W and v_0 is the corresponding eigenvector and $a_0(0)$ is the initial coefficient. This is because due to exponential growth, the largest eigenvalue will dominate the rate.

So for the random symmetric network, the decoding error increases. This is because the largest eigenvector which looks like noise will dominate the rate and so the decoding error will keep increasing as it looks less and less like the input feature vector $\mathbf{h}(\pi)$.

On the other hand the symmetric ring network's decoding error decreases. This is because the first 2 eigenvalues are similar in size and have the largest coefficients. So the combination of their eigenvectors will dominate. This combination looks basically the same as the input feature vector $\mathbf{h}(\pi)$ (other inputs should also work as the third eigenvector (sin wave instead of cos) also has a similar eigenvalue, but has a much smaller initial coefficient in this particular input). and so the decoding error will decrease as the decoding noise will becomes less and less important as the rate keeps increasing. One other thing to not for the symmetric ring is that the decoding error is very smooth when averaged over trials. Unlike the balanced ring network which has more noise.

Q7 Alternate B input matrix for balanced ring structure

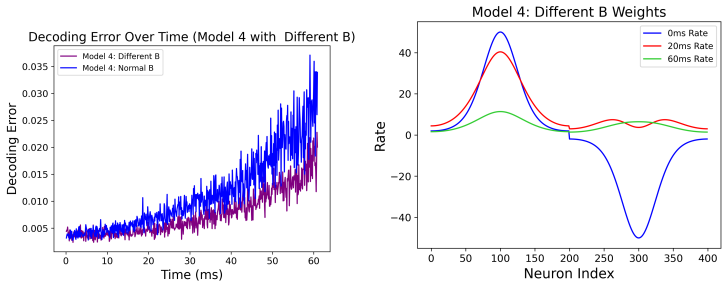

Figure 4: a) Decoding error for different B input matrice. b) Rate of neurons at $t = [0^+, 20ms, 60ms]$

One alternate B input matrix is for $B = [I_m - I_m]^T$. This initializes the inhibitory neurons with now a negative input instead of 0. This causes the decoding error to decrease even more than the original B input matrix as the negative input causes the excitatory neurons to be more excited and so the rate of the excitatory neurons is larger. This decrease in decoding error is shown in fig 4a and the rate of the neurons is shown in fig 4b. One thing to say here is that a negative rate is not biologically plausible. However if we say all these rates are changes from a baseline firing rate then this is fine.

Q8 Increasing α

Fig 3c shows the decoding error for the balanced ring network for different α values. It shows that the decoding error decreases as we increase α. However there is an issue with this. Specifically it causes the transient amplification of rate to become large and larger. In the real brain there is limits to the max firing rate of neurons. So if we keep increasing α the rate will eventually exceed this limit and the model will no longer be valid. Fig 3d and e shows this for excitatory neuron 100 and inhibitory neuron 300 for different α values.

Q9 Biological Implication of Results and Conclusion

Overall we find that in order for V1 to represent a specific feature in the face of readout noise it much have recurrent connectivity. As the zero network decays too quickly. However this recurrent connectivity can not be completely random like the random symmetric network as the decoding error will be too large and the rate will be too noisy (though actually a random uniform recurrent matrix distributed between 0 and 1 worked surprisingly well) due to poorly shaped eigenvectors.

Instead the recurrent connectivity could be a symmetric ring structure. In the symmetric ring network, recurrent connectivity is strongest between neurons with similar preferred orientations. This keeps the rate of the neurons similar to the input feature vector $\mathbf{h}(\pi)$ for longer and so the decoding error is small. However in this model inhibitory and excitatory neurons are modelled as the same neuron. In doing so we this decreases the amount of noise in the decoding error but also allows us to increase the eigenvalues of the recurrent connectivity matrix till it is greater than 1 which causes the rate to increase to infinity. This is not biologically plausible as the rate of neurons in the brain is limited.

Lastly there is the balanced ring network. This is the symmetric ring network but with the inhibitory neurons and excitatory neurons separated. In it we find a network which has a fast transient amplification of the rate (which is similar to V1 quick amplification of noise) but also in the long term the rate decays even when $\alpha > 1$. This is good and the decoding error when $\alpha = 5$ is lower than the symmetric ring network when $\alpha = 0.9$ so this model show cases an improvement. However one key issue is that increasing α causes the transient amplification to become larger and larger which cause the peak rate to keep increasing. One possible solution to this would be to add a saturating non-linearity to the model. This would cause the rate to saturate at a certain level and so the transient amplification would not be able to increase the rate above this level. However we would have to check that this still allows a good decoding performance.

Code

Simulation Class

```
import math
from enum import IntEnum

import numpy as np


class weight_options_enum(IntEnum):
    """
    Enum for weight options.
    """

    zeros = 0
    random_symmetric = 1
    symmetric_ring = 2
    balanced_ring = 3
    random_uniform_symmetric = 4
    non_normal_balanced_ring = 5


class V1_simulation:
    def __init__(
        self,
        number_of_orientations,
        kappa=1.0,
        number_of_neurons=100,
        weight_options=weight_options_enum.zeros,
        alpha=0.1,
        time_step=0.001,
        tau=0.1,
        sigma=1,
        diff_beta=False,
    ):
        """
        number_of_orientations: number of orientations to simulate (int)
        kappa: parameter for the Gaussian function (float)
        number_of_neurons: number of neurons in the simulation (int)
        """
        self.number_of_orientations = number_of_orientations
        self.kappa = kappa
        self.number_of_neurons = number_of_neurons
        self.orientations = (
            2
            * np.pi
            * np.arange(self.number_of_orientations)
            / self.number_of_orientations
        )
        self.alpha = alpha
        self.B_matrix = np.eye(self.number_of_neurons, self.number_of_orientations)
        if diff_beta:
            self.B_matrix = np.zeros(
```

```
                (self.number_of_neurons, self.number_of_orientations)
            )
            for i in range(self.number_of_orientations):
                """self.B_matrix[i, :] = np.exp(
                    (np.cos(self.orientations[i] - self.orientations) - 1)
                    / self.kappa**2
                )
                self.B_matrix[i + self.number_of_orientations, :] = -np.exp(
                    (np.cos(self.orientations[i] - self.orientations) - 1)
                    / self.kappa**2
                )"""
                """self.B_matrix[i, :] = 0.01
                self.B_matrix[i, i] = 1"""
                self.B_matrix[i, i] = 1
                self.B_matrix[i + self.number_of_orientations, i] = -1
            self.B_matrix = self.B_matrix / np.sum(self.B_matrix[0:200, :], axis=0)
        self.C_matrix = np.eye(self.number_of_orientations, self.number_of_neurons)

        self.time_step = time_step
        self.tau = tau
        self.sigma = sigma

        # Initialize the recurrent weights
        self.recurrent_weights = None
        self.weight_options = weight_options
        self.set_recurrent_weights(weight_options)

    def stimulus(self, left_orientation, right_orientation):
        # Simulate a stimulus based on the orientation
        stimulus_input = np.exp(
            (np.cos(left_orientation - right_orientation) - 1) / self.kappa**2
        )
        return stimulus_input

    def set_recurrent_weights(self, weight_options):
        """
        Set the recurrent weights based on the specified weight options.
        """
        if weight_options == weight_options_enum.zeros:
            assert self.number_of_neurons == self.number_of_orientations, (
                "The number of neurons must be equal to the number of orientations"
            )
            self.recurrent_weights = np.zeros(
                (self.number_of_neurons, self.number_of_neurons)
            )
        elif weight_options == weight_options_enum.random_symmetric:
            assert self.number_of_neurons == self.number_of_orientations, (
                "The number of neurons must be equal to the number of orientations"
            )
            self.recurrent_weights = np.random.randn(
                self.number_of_neurons, self.number_of_neurons
            )
            self.recurrent_weights = self.recurrent_weights + self.recurrent_weights.T
```

```
        self.recurrent_weights = self.set_weight_alpha(self.recurrent_weights)
    elif weight_options == weight_options_enum.random_uniform_symmetric:
        assert self.number_of_neurons == self.number_of_orientations, (
            "The number of neurons must be equal to the number of orientations"
        )
        self.recurrent_weights = (
            np.random.rand(self.number_of_neurons, self.number_of_neurons) - 0.5
        ) * 2
        self.recurrent_weights = self.recurrent_weights + self.recurrent_weights.T
        self.recurrent_weights = self.set_weight_alpha(self.recurrent_weights)
    elif weight_options == weight_options_enum.symmetric_ring:
        assert self.number_of_neurons == self.number_of_orientations, (
            "The number of neurons must be equal to the number of orientations"
        )
        self.recurrent_weights = np.zeros(
            (self.number_of_neurons, self.number_of_neurons)
        )
        for i in range(self.number_of_neurons):
            self.recurrent_weights[i, :] = self.stimulus(
                self.orientations[i], self.orientations
            )
        self.recurrent_weights = self.set_weight_alpha(self.recurrent_weights)
    elif weight_options == weight_options_enum.balanced_ring:
        assert self.number_of_neurons == 2 * self.number_of_orientations, (
            "The number of neurons must be 2* the number of orientations"
        )
        self.recurrent_weights = np.zeros(
            (self.number_of_neurons, self.number_of_neurons)
        )
        temp_matrix = np.zeros(
            (self.number_of_orientations, self.number_of_orientations)
        )
        for i in range(self.number_of_orientations):
            temp_matrix[i, :] = self.stimulus(
                self.orientations[i], self.orientations
            )
        temp_matrix = self.set_weight_alpha(temp_matrix)
        self.recurrent_weights = np.block(
            [[temp_matrix, -temp_matrix], [temp_matrix, -temp_matrix]]
        )
    elif weight_options == weight_options_enum.non_normal_balanced_ring:
        assert self.number_of_neurons == 2 * self.number_of_orientations, (
            "The number of neurons must be 2* the number of orientations"
        )
        self.recurrent_weights = np.zeros(
            (self.number_of_neurons, self.number_of_neurons)
        )
        temp_matrix = np.zeros(
            (self.number_of_orientations, self.number_of_orientations)
        )
        for i in range(self.number_of_orientations):
            temp_matrix[i, :] = self.stimulus(
                self.orientations[i], self.orientations
```

```
                )
            temp_matrix = self.set_weight_alpha(temp_matrix)
            self.recurrent_weights = np.block(
                [[temp_matrix, -1.2 * temp_matrix], [temp_matrix, -1.2 * temp_matrix]]
            )
        else:
            raise ValueError("Invalid weight options. Choose from weight_options_enum.")

    def set_alpha(self, alpha):
        """
        Set the alpha parameter.
        """
        self.alpha = alpha
        self.recurrent_weights = self.set_weight_alpha(self.recurrent_weights)

    def recreate_recurrent_weights(self, weight_options):
        """
        Recreate the recurrent weights with the specified weight options.
        """
        self.set_recurrent_weights(weight_options)
        self.weight_options = weight_options

    def set_weight_alpha(self, weight, alpha=None):
        current_alpha = self.alpha if alpha is None else alpha
        largest_real_eigenvalue = np.max(np.real(np.linalg.eigvals(weight)))
        scale = current_alpha / largest_real_eigenvalue
        weight = scale * weight
        assert math.isclose(
            np.max(np.real(np.linalg.eigvals(weight))), current_alpha, rel_tol=1e-5
        ), "The largest real eigenvalue is not equal to alpha"
        return weight

    def get_recurrent_weights(self):
        """
        Get the recurrent weights.
        """
        return self.recurrent_weights

    def get_stimulus(self, orientation):
        """
        Get the stimulus function.
        """
        return self.stimulus(self.orientations, orientation)

    def run_simulation(self, orientation, time):
        """
        Run the simulation for a given time with the specified orientations.
        """
        stimulus_input = self.stimulus(self.orientations, orientation)
        rate = np.zeros((self.number_of_neurons, int(time / self.time_step)))
        noisy_rate = np.zeros((self.number_of_orientations, int(time / self.time_step)))
        noisy_readout = np.zeros((1, int(time / self.time_step)))
```

```
            time_array = np.arange(0, time, self.time_step)

            time_index = 1

            rate[:, 1] = (self.B_matrix @ stimulus_input) / self.tau
            noisy_rate[:, 1] = self.C_matrix @ rate[:, 1] + self.sigma * np.random.randn(
                self.number_of_orientations
            )
            noisy_readout[:, 1] = np.atan2(
                np.sum(np.sin(self.orientations) * noisy_rate[:, 1]),
                np.sum(np.cos(self.orientations) * noisy_rate[:, 1]),
            )

            time_index = 2
            for i in range(2, int(time / self.time_step)):
                rate[:, i] = (
                    rate[:, i - 1]
                    + self.time_step
                    * (-rate[:, i - 1] + self.recurrent_weights @ rate[:, i - 1])
                    / self.tau
                )
                noisy_rate[:, i] = self.C_matrix @ rate[
                    :, i
                ] + self.sigma * np.random.randn(self.number_of_orientations)
                noisy_readout[:, i] = np.atan2(
                    np.sum(np.sin(self.orientations) * noisy_rate[:, i]),
                    np.sum(np.cos(self.orientations) * noisy_rate[:, i]),
                )
                time_index += 1
            return rate.T, noisy_rate.T, noisy_readout.T, time_array
```

Jupyter Notebook

```
%load_ext autoreload
%autoreload 2
import matplotlib.pyplot as plt
import numpy as np
import scipy

from main import V1_simulation, weight_options_enum

model_1 = V1_simulation(
    number_of_orientations=200,
    number_of_neurons=200,
    tau=20 * 1e-3,
    weight_options=weight_options_enum.zeros,
    kappa=np.pi / 4,
    alpha=0.9,
    sigma=1,
    time_step=1e-3,
)
model_2 = V1_simulation(
    number_of_orientations=200,
```

```
    number_of_neurons=200,
    tau=20 * 1e-3,
    weight_options=weight_options_enum.random_symmetric,
    kappa=np.pi / 4,
    alpha=0.9,
    sigma=1,
    time_step=1e-3,
)
model_2_uniform = V1_simulation(
    number_of_orientations=200,
    number_of_neurons=200,
    tau=20 * 1e-3,
    weight_options=weight_options_enum.random_uniform_symmetric,
    kappa=np.pi / 4,
    alpha=0.9,
    sigma=1,
    time_step=1e-3,
)
model_3 = V1_simulation(
    number_of_orientations=200,
    number_of_neurons=200,
    tau=20 * 1e-3,
    weight_options=weight_options_enum.symmetric_ring,
    kappa=np.pi / 4,
    alpha=0.9,
    sigma=1,
    time_step=1e-3,
)
model_4 = V1_simulation(
    number_of_orientations=200,
    number_of_neurons=400,
    tau=20 * 1e-3,
    weight_options=weight_options_enum.balanced_ring,
    kappa=np.pi / 4,
    alpha=0.9,
    sigma=1,
    time_step=1e-3,
)

colors = ["blue", "red", "limegreen"]
figure, ax = plt.subplots(1, 4, figsize=(22, 5))
figure.tight_layout(pad=0.4, w_pad=0.5, h_pad=1.0, rect=[0, 0.02, 1, 0.95])
figure.subplots_adjust(hspace=0.4, wspace=0.2)
figure.suptitle("Rate of Neurons at Different Time Steps", fontsize=16, y=1.0)

plt.subplot(1, 4, 1)
rate, noisy_rate, noisy_readout, time = model_1.run_simulation(
    orientation=np.pi, time=61 * 1e-3
)
rate = rate[1:]

thrity_ms = np.where(time == 0.02)[0][0] - 1
sixty_ms = np.where(time == 0.06)[0][0] - 1
```

```
print(thrity_ms)
plt.plot(rate[0], label="0ms Rate", color=colors[0])
plt.plot(rate[thrity_ms], label="20ms Rate", color=colors[1])
plt.plot(rate[sixty_ms], label="60ms Rate", color=colors[2])
plt.xlabel("Neuron Index", fontsize=16)
plt.ylabel("Rate", fontsize=16)
plt.title("Model 1: Zero Weights", fontsize=16)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.legend()


plt.subplot(1, 4, 2)

rate, noisy_rate, noisy_readout, time = model_2.run_simulation(
    orientation=np.pi, time=61 * 1e-3
)
rate = rate[1:]

thrity_ms = np.where(time == 0.02)[0][0] - 1
sixty_ms = np.where(time == 0.06)[0][0] - 1
print(thrity_ms)
plt.plot(rate[0], label="0ms Rate", color=colors[0])
plt.plot(rate[thrity_ms], label="20ms Rate", color=colors[1])
plt.plot(rate[sixty_ms], label="60ms Rate", color=colors[2])
plt.xlabel("Neuron Index", fontsize=16)
plt.ylabel("Rate", fontsize=16)
plt.title("Model 2: Zero Weights", fontsize=16)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.legend()

plt.subplot(1, 4, 3)

rate, noisy_rate, noisy_readout, time = model_3.run_simulation(
    orientation=np.pi, time=61 * 1e-3
)
rate = rate[1:]

thrity_ms = np.where(time == 0.02)[0][0] - 1
sixty_ms = np.where(time == 0.06)[0][0] - 1
print(thrity_ms)
plt.plot(rate[0], label="0ms Rate", color=colors[0])
plt.plot(rate[thrity_ms], label="20ms Rate", color=colors[1])
plt.plot(rate[sixty_ms], label="60ms Rate", color=colors[2])
plt.xlabel("Neuron Index", fontsize=16)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.ylabel("Rate", fontsize=16)
plt.title("Model 3: Zero Weights", fontsize=16)
plt.legend()

plt.subplot(1, 4, 4)
```

```
rate, noisy_rate, noisy_readout, time = model_4.run_simulation(
    orientation=np.pi, time=61 * 1e-3
)
rate = rate[1:]

thrity_ms = np.where(time == 0.02)[0][0] - 1
sixty_ms = np.where(time == 0.06)[0][0] - 1
print(thrity_ms)
plt.plot(rate[0], label="0ms Rate", color=colors[0])
plt.plot(rate[thrity_ms], label="20ms Rate", color=colors[1])
plt.plot(rate[sixty_ms], label="60ms Rate", color=colors[2])
plt.xlabel("Neuron Index", fontsize=16)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.ylabel("Rate", fontsize=16)
plt.title("Model 4: Zero Weights", fontsize=16)
plt.legend()

plt.show()
figure.savefig("rate_of_neurons.png", dpi=300, bbox_inches="tight")

# Q2, Q3
plt.rcParams.update({"font.size": 15})
figure, ax = plt.subplots(1, 5, figsize=(27, 7))
figure.tight_layout(pad=0.4, w_pad=0.5, h_pad=1.0, rect=[0, 0.02, 1, 0.95])
figure.subplots_adjust(hspace=0.4, wspace=0.2)
model_2_recurrent_weights = model_2.get_recurrent_weights()
model_3_recurrent_weights = model_3.get_recurrent_weights()
eigenvals = np.linalg.eigvals(model_2_recurrent_weights)
eigenvals = np.sort(eigenvals)[::-1]
model_3_eigenvals = np.linalg.eigvals(model_3_recurrent_weights)
model_3_eigenvals = np.sort(model_3_eigenvals)[::-1]

plt.subplot(1, 5, 1)
y = np.arange(len(eigenvals))
plt.scatter(y, eigenvals, label="Model 2: Random Weights Eigenvalues", c="b", alpha=0.5)
plt.scatter(
    y,
    model_3_eigenvals,
    label="Model 3: Symmetric Ring Weights Eigenvalues",
    c="r",
    alpha=0.5,
)
plt.title("Real Eigenvalues of Recurrent Weights")
plt.xlabel("Eigenvector Index")
plt.ylabel("Eigenvalue")
plt.legend(prop={"size": 12})

plt.subplot(1, 5, 2)
eigen_vals_model_2, eigen_vectors_model_2 = (
    np.linalg.eig(model_2_recurrent_weights)[0],
    np.linalg.eig(model_2_recurrent_weights)[1],
```

```
)
eigen_vals_model_3, eigen_vectors_model_3 = (
    np.linalg.eig(model_3_recurrent_weights)[0],
    np.linalg.eig(model_3_recurrent_weights)[1],
)
eigen_vals_model_4, eigen_vectors_model_4 = (
    np.linalg.eig(model_4.get_recurrent_weights())[0],
    np.linalg.eig(model_4.get_recurrent_weights())[1],
)
idx = np.argsort(np.abs(eigen_vals_model_2))[::-1]
eigen_vectors_model_2 = eigen_vectors_model_2[:, idx]
eigen_vals_model_2 = eigen_vals_model_2[idx]
idx = np.argsort(np.abs(eigen_vals_model_3))[::-1]
eigen_vectors_model_3 = eigen_vectors_model_3[:, idx]
eigen_vals_model_3 = eigen_vals_model_3[idx]
input = model_2.get_stimulus(np.pi)
decomposed = scipy.linalg.solve(eigen_vectors_model_2, input)
decomposed_model_3 = scipy.linalg.solve(eigen_vectors_model_3, input)


eigen_vector_index = np.arange(len(decomposed))
plt.scatter(
    eigen_vector_index, decomposed, label="Model 2: Random Weights", c="b", alpha=0.7
)
plt.scatter(
    eigen_vector_index,
    decomposed_model_3,
    label="Model 3: Symmetric Ring Weights",
    c="r",
    alpha=0.7,
)
plt.title("Decomposed Input into Eigenvectors")
plt.xlabel("Eigenvector Index")
plt.ylabel("Decomposed Coefficient")
plt.legend(prop={"size": 12})

plt.subplot(1, 5, 3)

eigen_vals_model_2, eigen_vectors_model_2 = (
    np.linalg.eig(model_2_recurrent_weights)[0],
    np.linalg.eig(model_2_recurrent_weights)[1],
)
eigen_vals_model_3, eigen_vectors_model_3 = (
    np.linalg.eig(model_3_recurrent_weights)[0],
    np.linalg.eig(model_3_recurrent_weights)[1],
)

largest_eigen_val_model_2 = np.max(np.abs(eigen_vals_model_2))
largest_eigen_val_model_3 = np.max(np.abs(eigen_vals_model_3))
largest_eigen_vec_model_2 = eigen_vectors_model_2[
    :, np.argmax(np.abs(eigen_vals_model_2))
]
largest_eigen_vec_model_3 = eigen_vectors_model_3[
```

```
    :, np.argmax(np.abs(eigen_vals_model_3))
]
plt.plot(largest_eigen_vec_model_2, label="Model 2: Random Weights Eigenvector", c="b")
plt.plot(
    largest_eigen_vec_model_3,
    label="Model 3: Symmetric Ring Weights Eigenvector",
    c="r",
)
plt.title("Eigenvector of Largest Eigenvalue")
plt.xlabel("Neuron Index")
plt.ylabel("Eigenvector Value")
plt.legend(prop={"size": 12})

plt.subplot(1, 5, 4)
second_largest_eigen_val_model_2 = np.sort(np.abs(eigen_vals_model_2))[-2]
second_largest_eigen_val_model_3 = np.sort(np.abs(eigen_vals_model_3))[-2]
second_largest_eigen_vec_model_2 = eigen_vectors_model_2[
    :, np.argsort(np.abs(eigen_vals_model_2))[-2]
]
second_largest_eigen_vec_model_3 = eigen_vectors_model_3[
    :, np.argsort(np.abs(eigen_vals_model_3))[-2]
]
plt.plot(
    second_largest_eigen_vec_model_2, label="Model 2: Random Weights Eigenvector", c="b"
)
plt.plot(
    second_largest_eigen_vec_model_3,
    label="Model 3: Symmetric Ring Weights Eigenvector",
    c="r",
)
plt.title("Eigenvector of Second Largest Eigenvalue")
plt.xlabel("Neuron Index")
plt.ylabel("Eigenvector Value")
plt.legend(prop={"size": 12})

plt.subplot(1, 5, 5)

rate, noisy_rate, noisy_readout, time = model_2.run_simulation(
    orientation=np.pi, time=61 * 1e-3
)
rate = rate[1:]

twenty_ms = np.where(time == 0.02)[0][0] - 1
twenty_five_ms = np.where(time == 0.025)[0][0] - 1
thirty_ms = np.where(time == 0.03)[0][0] - 1
print(thrity_ms)
plt.plot(rate[twenty_ms], label="20ms Rate", color=colors[0])
plt.plot(rate[twenty_five_ms], label="25ms Rate", color=colors[1])
plt.plot(rate[thirty_ms], label="30ms Rate", color=colors[2])
plt.xlabel("Neuron Index", fontsize=16)
plt.ylabel("Rate", fontsize=16)
plt.title("Model 2: Random Weights", fontsize=16)
plt.legend()
```

```
plt.savefig("eigenvalues.png", dpi=300, bbox_inches="tight")
plt.show()

# Q4
averages = 100
model_1_decoding_error = np.zeros((60, averages))
model_2_decoding_error = np.zeros((60, averages))
model_3_decoding_error = np.zeros((60, averages))
model_4_decoding_error = np.zeros((60, averages))
for i in range(averages):
    model_1_rate, model_1_noisy_rate, model_1_noisy_readout, model_1_time = (
        model_1.run_simulation(orientation=np.pi, time=61 * 1e-3)
    )
    model_1_rate = model_1_rate[1:]
    model_1_noisy_rate = model_1_noisy_rate[1:]
    model_1_noisy_readout = model_1_noisy_readout[1:]
    model_1_decoding_error[:, i] = np.acos(
        np.cos(model_1_noisy_readout - np.pi)
    ).flatten()

for i in range(averages):
    model_2.recreate_recurrent_weights(weight_options_enum.random_symmetric)
    model_2_rate, model_2_noisy_rate, model_2_noisy_readout, model_2_time = (
        model_2.run_simulation(orientation=np.pi, time=61 * 1e-3)
    )
    model_2_rate = model_2_rate[1:]
    model_2_noisy_rate = model_2_noisy_rate[1:]
    model_2_noisy_readout = model_2_noisy_readout[1:]
    model_2_decoding_error[:, i] = np.acos(
        np.cos(model_2_noisy_readout - np.pi)
    ).flatten()

for i in range(averages):
    model_3_rate, model_3_noisy_rate, model_3_noisy_readout, model_3_time = (
        model_3.run_simulation(orientation=np.pi, time=61 * 1e-3)
    )
    model_3_rate = model_3_rate[1:]
    model_3_noisy_rate = model_3_noisy_rate[1:]
    model_3_noisy_readout = model_3_noisy_readout[1:]
    model_3_decoding_error[:, i] = np.acos(
        np.cos(model_3_noisy_readout - np.pi)
    ).flatten()

for i in range(averages):
    model_4_rate, model_4_noisy_rate, model_4_noisy_readout, model_4_time = (
        model_4.run_simulation(orientation=np.pi, time=61 * 1e-3)
    )
    model_4_rate = model_4_rate[1:]
    model_4_noisy_rate = model_4_noisy_rate[1:]
    model_4_noisy_readout = model_4_noisy_readout[1:]
    model_4_decoding_error[:, i] = np.acos(
        np.cos(model_4_noisy_readout - np.pi)
```

```
    ).flatten()

model_1_decoding_error = np.mean(model_1_decoding_error, axis=1)
model_2_decoding_error = np.mean(model_2_decoding_error, axis=1)
model_3_decoding_error = np.mean(model_3_decoding_error, axis=1)
model_4_decoding_error = np.mean(model_4_decoding_error, axis=1)

plt.rcParams.update({"font.size": 11})
colors = ["blue", "red", "limegreen", "purple"]
plt.plot(
    model_1_time[1:] / 1e-3,
    model_1_decoding_error,
    label="Model 1: Zero Weights",
    color=colors[0],
)
plt.plot(
    model_2_time[1:] / 1e-3,
    model_2_decoding_error,
    label="Model 2: Random Weights",
    color=colors[1],
)
plt.plot(
    model_3_time[1:] / 1e-3,
    model_3_decoding_error,
    label="Model 3: Symmetric Ring Weights",
    color=colors[2],
)
plt.plot(
    model_4_time[1:] / 1e-3,
    model_4_decoding_error,
    label="Model 4: Balanced Ring Weights",
    color=colors[3],
)
plt.xlabel("Time (ms)")
plt.ylabel("Decoding Error")
plt.title("Decoding Error Over Time")
plt.legend()
plt.savefig("error.png", dpi=300, bbox_inches="tight")
plt.show()

model_4_alpha_5 = V1_simulation(
    number_of_orientations=200,
    number_of_neurons=400,
    tau=20 * 1e-3,
    weight_options=weight_options_enum.balanced_ring,
    kappa=np.pi / 4,
    alpha=5,
    sigma=1,
    time_step=1e-4,
)
model_2_alpha_5 = V1_simulation(
    number_of_orientations=200,
    number_of_neurons=200,
```

```
    tau=20 * 1e-3,
    weight_options=weight_options_enum.random_symmetric,
    kappa=np.pi / 4,
    alpha=5,
    sigma=1,
    time_step=1e-4,
)
model_3_alpha_5 = V1_simulation(
    number_of_orientations=200,
    number_of_neurons=200,
    tau=20 * 1e-3,
    weight_options=weight_options_enum.symmetric_ring,
    kappa=np.pi / 4,
    alpha=5,
    sigma=1,
    time_step=1e-4,
)

averages = 20
model_2_decoding_error_alpha_5 = np.zeros((609, averages))
model_3_decoding_error_alpha_5 = np.zeros((609, averages))
model_4_decoding_error_alpha_5 = np.zeros((609, averages))

for ii in range(averages):
    for i in range(2, 5):
        model = globals()[f"model_{i}_alpha_5"]
        if i == 2:
            model.recreate_recurrent_weights(weight_options_enum.random_symmetric)
        model_rate, model_noisy_rate, model_noisy_readout, model_time = (
            model.run_simulation(orientation=np.pi, time=61 * 1e-3)
        )
        model_rate = model_rate[1:]
        model_noisy_rate = model_noisy_rate[1:]
        model_noisy_readout = model_noisy_readout[1:]
        model_decoding_error = np.acos(np.cos(model_noisy_readout - np.pi)).flatten()
        globals()[f"model_{i}_decoding_error_alpha_5"][:, ii] = model_decoding_error

model_4_decoding_error_alpha_5 = np.zeros((609, averages))
plt.plot(model_rate[0], label="0ms Rate", color=colors[0])
plt.plot(model_rate[20], label="20ms Rate", color=colors[1])
plt.plot(model_rate[60], label="60ms Rate", color=colors[2])
plt.xlabel("Neuron Index", fontsize=16)
plt.ylabel("Rate", fontsize=16)
plt.title("Model 4: Balanced Ring Weights", fontsize=16)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.legend()
plt.show()

# Q5 and Q6
model_2_decoding_error_alpha_5_mean = np.mean(model_2_decoding_error_alpha_5, axis=1)
model_3_decoding_error_alpha_5_mean = np.mean(model_3_decoding_error_alpha_5, axis=1)
model_4_decoding_error_alpha_5_mean = np.mean(model_4_decoding_error_alpha_5, axis=1)
```

```
time = np.arange(1e-4, 61 * 1e-3, 1e-4)
plt.plot(
    time,
    model_2_decoding_error_alpha_5_mean,
    label="Model 2: Random Weights",
    color=colors[1],
)
plt.plot(
    time,
    model_3_decoding_error_alpha_5_mean,
    label="Model 3: Symmetric Ring Weights",
    color=colors[2],
)
plt.plot(
    time,
    model_4_decoding_error_alpha_5_mean,
    label="Model 4: Balanced Ring Weights",
    color=colors[3],
)
plt.xlabel("Time (s)")
plt.ylabel("Decoding Error")

axins = plt.gca().inset_axes(
    [0.2, 0.2, 0.79, 0.2], ylim=(0, 0.01), xlim=(0, 0.062)
)  # [x0, y0, width, height]
axins.plot(time, model_2_decoding_error_alpha_5_mean, color=colors[1])
axins.plot(time, model_3_decoding_error_alpha_5_mean, color=colors[2])
axins.plot(time, model_4_decoding_error_alpha_5_mean, color=colors[3])
plt.gca().indicate_inset_zoom(axins)
plt.title("Decoding Error Over Time (Alpha = 5)")
plt.legend()
plt.savefig("error_alpha_5.png", dpi=300, bbox_inches="tight")
plt.show()

model_4_alpha_03 = V1_simulation(
    number_of_orientations=200,
    number_of_neurons=400,
    tau=20 * 1e-3,
    weight_options=weight_options_enum.balanced_ring,
    kappa=np.pi / 4,
    alpha=0.3,
    sigma=1,
    time_step=1e-4,
)
model_4_alpha_09 = V1_simulation(
    number_of_orientations=200,
    number_of_neurons=400,
    tau=20 * 1e-3,
    weight_options=weight_options_enum.balanced_ring,
    kappa=np.pi / 4,
    alpha=0.9,
    sigma=1,
    time_step=1e-4,
```

```
)
model_4_alpha_2 = V1_simulation(
    number_of_orientations=200,
    number_of_neurons=400,
    tau=20 * 1e-3,
    weight_options=weight_options_enum.balanced_ring,
    kappa=np.pi / 4,
    alpha=2,
    sigma=1,
    time_step=1e-4,
)
model_4_alpha_5 = V1_simulation(
    number_of_orientations=200,
    number_of_neurons=400,
    tau=20 * 1e-3,
    weight_options=weight_options_enum.balanced_ring,
    kappa=np.pi / 4,
    alpha=5,
    sigma=1,
    time_step=1e-4,
)
model_4_alpha_10 = V1_simulation(
    number_of_orientations=200,
    number_of_neurons=400,
    tau=20 * 1e-3,
    weight_options=weight_options_enum.balanced_ring,
    kappa=np.pi / 4,
    alpha=10,
    sigma=1,
    time_step=1e-4,
)
model_4_alpha_20 = V1_simulation(
    number_of_orientations=200,
    number_of_neurons=400,
    tau=20 * 1e-3,
    weight_options=weight_options_enum.balanced_ring,
    kappa=np.pi / 4,
    alpha=20,
    sigma=1,
    time_step=1e-4,
)

# Q8
averages = 20
model_4_decoding_error_alpha_03 = np.zeros((1209, averages))
model_4_rate_alpha_03 = np.zeros((1209, 400, averages))
model_4_decoding_error_alpha_09 = np.zeros((1209, averages))
model_4_rate_alpha_09 = np.zeros((1209, 400, averages))
model_4_decoding_error_alpha_2 = np.zeros((1209, averages))
model_4_rate_alpha_2 = np.zeros((1209, 400, averages))
model_4_decoding_error_alpha_5 = np.zeros((1209, averages))
model_4_rate_alpha_5 = np.zeros((1209, 400, averages))
model_4_decoding_error_alpha_10 = np.zeros((1209, averages))
```

```
model_4_rate_alpha_10 = np.zeros((1209, 400, averages))
model_4_decoding_error_alpha_20 = np.zeros((1209, averages))
model_4_rate_alpha_20 = np.zeros((1209, 400, averages))
for ii in range(averages):
    for i in ["03", "09", "2", "5", "10", "20"]:
        model = globals()[f"model_4_alpha_{i}"]
        model_rate, model_noisy_rate, model_noisy_readout, model_time = (
            model.run_simulation(orientation=np.pi, time=121 * 1e-3)
        )
        model_rate = model_rate[1:]
        model_noisy_rate = model_noisy_rate[1:]
        model_noisy_readout = model_noisy_readout[1:]
        model_decoding_error = np.acos(np.cos(model_noisy_readout - np.pi)).flatten()
        globals()[f"model_4_decoding_error_alpha_{i}"][:, ii] = model_decoding_error
        globals()[f"model_4_rate_alpha_{i}"][:, :, ii] = model_rate

model_4_decoding_error_alpha_03_mean = np.mean(model_4_decoding_error_alpha_03, axis=1)
model_4_decoding_error_alpha_09_mean = np.mean(model_4_decoding_error_alpha_09, axis=1)
model_4_decoding_error_alpha_2_mean = np.mean(model_4_decoding_error_alpha_2, axis=1)
model_4_decoding_error_alpha_5_mean = np.mean(model_4_decoding_error_alpha_5, axis=1)
model_4_decoding_error_alpha_10_mean = np.mean(model_4_decoding_error_alpha_10, axis=1)
model_4_decoding_error_alpha_20_mean = np.mean(model_4_decoding_error_alpha_20, axis=1)

model_4_rate_alpha_03_mean = np.mean(model_4_rate_alpha_03, axis=2)
model_4_rate_alpha_09_mean = np.mean(model_4_rate_alpha_09, axis=2)
model_4_rate_alpha_2_mean = np.mean(model_4_rate_alpha_2, axis=2)
model_4_rate_alpha_5_mean = np.mean(model_4_rate_alpha_5, axis=2)
model_4_rate_alpha_10_mean = np.mean(model_4_rate_alpha_10, axis=2)
model_4_rate_alpha_20_mean = np.mean(model_4_rate_alpha_20, axis=2)

time = np.arange(1e-4, 121 * 1e-3, 1e-4)
time = time[:1209]
plt.plot(time, model_4_decoding_error_alpha_03_mean, label="Model 4: Alpha = 0.3")
plt.plot(time, model_4_decoding_error_alpha_09_mean, label="Model 4: Alpha = 0.9")
plt.plot(time, model_4_decoding_error_alpha_2_mean, label="Model 4: Alpha = 2")
plt.plot(time, model_4_decoding_error_alpha_5_mean, label="Model 4: Alpha = 5")
plt.plot(time, model_4_decoding_error_alpha_10_mean, label="Model 4: Alpha = 10")
plt.plot(time, model_4_decoding_error_alpha_20_mean, label="Model 4: Alpha = 20")
plt.xlabel("Time (s)")
plt.ylabel("Decoding Error")
plt.title("Decoding Error Over Time (Model 4)")
plt.legend()
plt.show()

model_4_rate_alpha_03_2 = np.zeros((5009, 400))
(
    model_4_rate_alpha_03_2,
    model_4_noisy_rate_alpha_03_2,
    model_4_noisy_readout_alpha_03_2,
    model_4_time_alpha_03_2,
) = model_4_alpha_03.run_simulation(orientation=np.pi, time=5009 * 1e-3)
model_4_rate_alpha_09_2 = np.zeros((5009, 400))
(
```

```
    model_4_rate_alpha_09_2,
    model_4_noisy_rate_alpha_09_2,
    model_4_noisy_readout_alpha_09_2,
    model_4_time_alpha_09_2,
) = model_4_alpha_09.run_simulation(orientation=np.pi, time=5009 * 1e-3)
model_4_rate_alpha_2_2 = np.zeros((5009, 400))
(
    model_4_rate_alpha_2_2,
    model_4_noisy_rate_alpha_2_2,
    model_4_noisy_readout_alpha_2_2,
    model_4_time_alpha_2_2,
) = model_4_alpha_2.run_simulation(orientation=np.pi, time=5009 * 1e-3)
model_4_rate_alpha_5_2 = np.zeros((5009, 400))
(
    model_4_rate_alpha_5_2,
    model_4_noisy_rate_alpha_5_2,
    model_4_noisy_readout_alpha_5_2,
    model_4_time_alpha_5_2,
) = model_4_alpha_5.run_simulation(orientation=np.pi, time=5009 * 1e-3)
model_4_rate_alpha_10_2 = np.zeros((5009, 400))
(
    model_4_rate_alpha_10_2,
    model_4_noisy_rate_alpha_10_2,
    model_4_noisy_readout_alpha_10_2,
    model_4_time_alpha_10_2,
) = model_4_alpha_10.run_simulation(orientation=np.pi, time=5009 * 1e-3)
model_4_rate_alpha_20_2 = np.zeros((5009, 400))
(
    model_4_rate_alpha_20_2,
    model_4_noisy_rate_alpha_20_2,
    model_4_noisy_readout_alpha_20_2,
    model_4_time_alpha_20_2,
) = model_4_alpha_20.run_simulation(orientation=np.pi, time=5009 * 1e-3)

time = np.arange(1e-4, 50090 * 1e-3, 1e-4)
time = time[:2008]
colors = ["blue", "red", "limegreen", "purple", "orange", "cyan"]
print(model_4_rate_alpha_03_2.shape)
print(model_4_rate_alpha_09_2.shape)
plt.plot(
    time, model_4_rate_alpha_03_2[1:2009, 100], label="Alpha = 0.3", color=colors[0]
)
plt.plot(
    time, model_4_rate_alpha_09_2[1:2009, 100], label="Alpha = 0.9", color=colors[1]
)
plt.plot(time, model_4_rate_alpha_2_2[1:2009, 100], label="Alpha = 2", color=colors[2])
plt.plot(time, model_4_rate_alpha_5_2[1:2009, 100], label="Alpha = 5", color=colors[3])
plt.plot(
    time, model_4_rate_alpha_10_2[1:2009, 100], label="Alpha = 10", color=colors[4]
)
plt.plot(
    time, model_4_rate_alpha_20_2[1:2009, 100], label="Alpha = 20", color=colors[5]
)
```

```
plt.xlabel("Time (s)")
plt.ylabel("Rate")
plt.title("Model 4: Rate for Neuron 100")
plt.legend()
plt.savefig("model_4_rate_alpha.png", dpi=300, bbox_inches="tight")
plt.show()

time = np.arange(1e-4, 50090 * 1e-3, 1e-4)
time = time[:2008]
colors = ["blue", "red", "limegreen", "purple", "orange", "cyan"]
print(model_4_rate_alpha_03_2.shape)
print(model_4_rate_alpha_09_2.shape)
plt.plot(
    time, model_4_rate_alpha_03_2[1:2009, 300], label="Alpha = 0.3", color=colors[0]
)
plt.plot(
    time, model_4_rate_alpha_09_2[1:2009, 300], label="Alpha = 0.9", color=colors[1]
)
plt.plot(time, model_4_rate_alpha_2_2[1:2009, 300], label="Alpha = 2", color=colors[2])
plt.plot(time, model_4_rate_alpha_5_2[1:2009, 300], label="Alpha = 5", color=colors[3])
plt.plot(
    time, model_4_rate_alpha_10_2[1:2009, 300], label="Alpha = 10", color=colors[4]
)
plt.plot(
    time, model_4_rate_alpha_20_2[1:2009, 300], label="Alpha = 20", color=colors[5]
)

plt.xlabel("Time (s)")
plt.ylabel("Rate")
plt.title("Model 4: Rate for Neuron 300")
plt.legend()
plt.savefig("model_4_rate_alpha_300.png", dpi=300, bbox_inches="tight")
plt.show()

# Q7
model_4_diff_beta = V1_simulation(
    number_of_orientations=200,
    number_of_neurons=400,
    tau=20 * 1e-3,
    weight_options=weight_options_enum.balanced_ring,
    kappa=np.pi / 4,
    alpha=0.9,
    sigma=1,
    time_step=1e-4,
    diff_beta=True,
)
model_4 = V1_simulation(
    number_of_orientations=200,
    number_of_neurons=400,
    tau=20 * 1e-3,
    weight_options=weight_options_enum.balanced_ring,
    kappa=np.pi / 4,
```

```
    alpha=0.9,
    sigma=1,
    time_step=1e-4,
)

averages = 20
model_4_diff_beta_decoding_error = np.zeros((609, averages))
model_4_decoding_error = np.zeros((609, averages))

for ii in range(averages):
    (
        model_4_diff_beta_rate,
        model_4_diff_beta_noisy_rate,
        model_4_diff_beta_noisy_readout,
        model_4_diff_beta_time,
    ) = model_4_diff_beta.run_simulation(orientation=np.pi, time=61 * 1e-3)
    model_4_diff_beta_rate = model_4_diff_beta_rate[1:]
    model_4_diff_beta_noisy_rate = model_4_diff_beta_noisy_rate[1:]
    model_4_diff_beta_noisy_readout = model_4_diff_beta_noisy_readout[1:]
    model_4_diff_beta_decoding_error[:, ii] = np.acos(
        np.cos(model_4_diff_beta_noisy_readout - np.pi)
    ).flatten()

for ii in range(averages):
    (
        model_4_rate,
        model_4_noisy_rate,
        model_4_noisy_readout,
        model_4_time,
    ) = model_4.run_simulation(orientation=np.pi, time=61 * 1e-3)
    model_4_rate = model_4_rate[1:]
    model_4_noisy_rate = model_4_noisy_rate[1:]
    model_4_noisy_readout = model_4_noisy_readout[1:]
    model_4_decoding_error[:, ii] = np.acos(
        np.cos(model_4_noisy_readout - np.pi)
    ).flatten()

model_4_diff_beta_decoding_error = np.mean(model_4_diff_beta_decoding_error, axis=1)
model_4_decoding_error = np.mean(model_4_decoding_error, axis=1)

plt.plot(
    model_4_diff_beta_time[1:] / 1e-3,
    model_4_diff_beta_decoding_error,
    label="Model 4: Different B",
    color="purple",
)
plt.plot(
    model_4_time[1:] / 1e-3,
    model_4_decoding_error,
    label="Model 4: Normal B",
    color="blue",
)
plt.xlabel("Time (ms)", fontsize=16)
```

```
plt.ylabel("Decoding Error", fontsize=16)
plt.title("Decoding Error Over Time (Model 4 with  Different B)", fontsize=16)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.legend()
plt.savefig("error_diff_beta.png", dpi=300, bbox_inches="tight")
plt.show()

plt.plot(model_4_diff_beta_rate[0], label="0ms Rate", color=colors[0])
plt.plot(model_4_diff_beta_rate[200], label="20ms Rate", color=colors[1])
plt.plot(model_4_diff_beta_rate[600], label="60ms Rate", color=colors[2])
plt.xlabel("Neuron Index", fontsize=16)
plt.ylabel("Rate", fontsize=16)
plt.title("Model 4: Different B Weights", fontsize=16)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.legend()
plt.savefig("error_diff_beta_rate.png", dpi=300, bbox_inches="tight")
plt.show()
```