Aide pour le projet Spiking SOM

1 Valeur des paramètres

Si vous n'arrivez pas à reproduire des comportements présentés dans l'article, vous pouvez faire varier les valeurs de paramètres.
N'initialisez aléatoirement que les poids synaptiques entre u et v, fixez les autres poids synaptiques à une valeur constante.

2 Calcul de distances toriques

L'espace d'entrée (des données) ou l'espace de sortie (la grille de neurones en deux dimensions de la SOM) est torique. Ainsi, nous avons besoin de calculer une distance dans un espace torique. Par exemple dans une grille torique dont l'intervalle de variation est compris entre 0 et 1, 1 est plus proche de 0 que de 0,8.
Ce calcul de distance torique est nécessaire à la fois pour l'encodage du vecteur d'entrée en tension pour la population de neurones u et à la fois pour implémenter le noyau d'interaction au sein de la population v.
Voici une fonction permettant de calculer une distance toroidale, vous pourrez l'adapter à vos besoins :

```
import numpy as np

def toroidalDistance(x1,x2):
    dx = np.abs(x1 - x2)
    tor_dist = np.where(dx < 0.5, dx, 1.0 - dx)
    return tor_dist
```

3 Implémentation des fonctions alphas

Les auteurs de l'article utilisent une variable binaire s représentant la présence ou l'absence instantanée de spike présynaptique. Cette formulation n'est pas implémentable telle quelle avec le simulateur Brian. En effet, on ne peut pas remettre s instantanément à 0 après l'émission d'un spike présynaptique.
Voici une solution potentielle pour implémenter les fonctions alpha sous Brian :

On introduit les fonctions alpha dans un modèle de neurone, qu'on appellera ici *u_layer_neuron_equ* :

```
u_layer_neuron_equ = '''
        I_ext : 1

        # inhibitory synapses to u layer : alpha functions
        ds_inh2u/dt = (-s_inh2u)/tau_r_inh2u: 1
        dI_inh2u/dt = (s_inh2u - I_inh2u)/tau_f_inh2u: 1

        # membrane potential of u layer
        dv/dt = (-v + I_ext - I_inh2u) / tau_m: 1
    '''
```

Ensuite lors de l'émission d'un spike présynaptique (en entrée d'un neurone) de *u_layer*, on incrémente *s_inh2u* du poids synaptique *w_syn* :

```
model_synapse_inh2u_inhibition = '''
w_syn : 1
'''

on_pre_synapse_inh2u_inhibition = '''
s_inh2u += w_syn
'''
```

Pour plus d'informations, consultez `https://brian2.readthedocs.io/en/stable/user/converting_from_integrated_form.html`.

4 Implémentation de la règle d'apprentissage STDP

Pour implémenter la règle STDP, souvenez-vous que Brian travaille avec des équations différentielles et des spikes. Nous avons vu dans le deuxième tutoriel sur les synapses comment reformuler la règle STDP en équations différentielles. Il vous suffit donc d'adapter la formulation que nous avons vu à la variante de STDP présentée dans l'article.

5 Implémentation du noyau d'interaction

Le noyau d'interaction prends la forme d'une différence de gaussienne implémentée via les poids synaptiques latéraux de la population de neurones v. Les neurones de v sont définis sur une grille torique en deux dimensions. Rappelons à cet endroit que les distances utilisées pour le calcul des poids synaptiques doivent prendre en compte la structure torique de v.
Le noyau d'interaction est implémenté avec la fonction suivante :

$$w_{i,j} = (1+a)\exp\left(-\frac{1}{2}\frac{d(i,j)^2}{r^2}\right) - a\exp\left(-\frac{1}{2}\frac{d(i,j)^2}{(br)^2}\right)$$

avec $d(i,j)$ étant la distance torique entre deux neurones sur la grille, e.g. la distance entre le neurone avec la position (0,0) et le neurone avec la position (5,2).
La fonction *r(t)* utilisée pour faire décroître le rayon d'interaction n'est pas nécessaire pour les tests que vous devez réaliser. Ainsi il suffit de calculer une fois les poids synaptiques latéraux de v et de les fixer.
Notez que sous Brian, toute population de neurones ou de synapses est définie en une dimension. Les fonctions *reshape* et *flatten* de numpy vous seront utiles pour changer la forme de vos tableaux.

6 Lancer une simulation Brian avec un jeu de données

On utilisera *store()* et *restore()* pour respectivement stocker et restaurer l'état du réseau. Ces fonctions sont utiles pour s'interfacer avec du code python permettant de présenter de façon itérative l'ensemble des vecteurs de notre jeu de données.

```
net_model = Network(collect())
net_model.store()

for epoch in range (nb_epoch):
    np.random.shuffle(dataset)
    for vector in enumerate(dataset) :
        for oscil in range(nb_oscil):
            net_model.restore()
            potential_input = rf.float_to_membrane_potential(vector)
            u_layer.I_ext = potential_input.flatten()
            net_model.run(oscillation_period)
            net_model.store()
```

7 Visualisation des valeurs préférentielles des neurones de la SOM

Pour vérifier qualitativement que votre apprentissage se déroule correctement, vous devez au préalable trouver les valeurs préférentielles des neurones de la SOM. Pour une SOM de 10*10 et des données en deux dimensions, cela résulte en un tableau numpy de la forme (10,10,2). Par exemple le neurone à la position (4,4) peut-être réglé pour réagir préférentiellement au motif [0.95, 0.05].
Comment trouver ces valeurs préférentielles ? Si on reprends l'exemple précédent, le neurone à la position (4,4) a 10 synapses qui le relient à une banque de 10 neurones au sein de u, représentant une dimension. En trouvant la synapse avec le poids le plus élevé, on peut retrouver la valeur préférentielle (pour une dimension) de ce neurone.

```
# dim : input dimension
# y : neuron position following y axis
# x : neuron position following x axis
index = np.argmax(weights[dim,:,y,x])
```

Pour obtenir une représentation plus facilement manipulable des poids synaptiques, vous pouvez adapter le code suivant (source : `https://brian2.readthedocs.io/en/stable/user/synapses.html#id9`) :

```
synapses = Synapses(source_group, target_group,
                    '''...
                       w : 1  # synaptic weight''', ...)
# ...
# Run the simulation
run(...)
# Create a matrix to store the weights and fill it with NaN
W = np.full((len(source_group), len(target_group)), np.nan)
# Insert the values from the Synapses object
W[synapses.i[:], synapses.j[:]] = synapses.w[:]
```

Pour visualiser le processus d'apprentissage, vous pouvez utiliser la classe *vizualization*. Le jeu de donnée que vous utilisez est demandé pour initialiser la classe. Ensuite, il faut appeler la fonction *update* en lui fournissant le vecteur d'entrée courant et le tableau numpy associant à chaque neurone de la SOM un vecteur de valeurs préférentielles. Cela résulte en un tableau numpy de forme (10,10,2) pour une SOM de 10*10 neurones avec des données en deux dimensions. Enfin, vous pouvez éventuellement passer en argument un titre à afficher.