Problème IA WEIERSTRASS

full source code on github.

Pour voir l'application de mon code, il faut aller sur le fichier WeierStrass.ipynb

Questions

Quelle est la taille de l'espace de recherche ?

L'espace de recherche est $]0,\ 1[\times [[1,\ 20]]^2$.

Aussi, si on décide d'arrondir **a** à 1e-2 près par exemple, on peut quantifier le nombre de possibilité. Soit le nombre de possibilités **n** et la précision de a **ε**. On a :

*n = 400 * (1 / ε - 1)*

Avec un précision de 1e-2, on a donc ~4000 possibilités. Il faut donc une petite population avec peu de cycle.

Quelle est votre fonction fitness ?

Etant donné que nous pouvons directement calculer la température avec un tuple (a, b, c) donnée, il nous suffit de calculer la température correspondant à l'individu et do sommer les différences en valeur absolu avec les sorties attendus :

```
def temperatureCalculIndividu(i, a, b, c):
    sum = .0
    ai = 1
    bi = 1
    ipi = i * math.pi
    cint = int(c)
    for n in range(cint + 1):
        sum += ai * math.cos(ipi * bi)
        if n == cint:
            break
        ai *= a
        bi *= b
    return sum

def fitnessCalculIndividu(a, b, c, time, temperature):
    cost = .0
    for i in range(len(time)):
        cost += abs(temperatureCalculIndividu(time[i], a, b, c) - temperature[i])
    return cost / len(time)
```

Cependant, ici on calcul les températures sur les individus 1 à 1. Mon amélioration a été de calculer la température sur la population entière. Le temps d'éxecution d'une génération a été divisé par presque 10. Cette amélioration n'est possible qu'avec une population suffisament grande (numpy calcul des cosinus avec ressemblance...) :

```
def temperatureCalcul(i, population):
    size = len(population)
    sum, ai, bi = np.zeros(size), np.ones(size), np.ones(size)
    ipi = i * math.pi
    a, b, c = population[:,0], population[:,1], population[:,2]
    cint = int(max(c))
    test = np.ones(size, dtype=bool)
    for n in range(cint + 1):
        temp = ai * np.cos(ipi * bi)
        test = np.logical_and(test, n <= c)
        temp *= test.astype(int)
        sum += temp
        if n == cint:
            break
        ai = ai * a * test.astype(int)
        bi = bi * b * test.astype(int)
    return sum

def total_fitness(population, time, temperature):
    cost = np.zeros(len(population))
    for i in range(len(time)):
        cost += np.absolute(temperatureCalcul(time[i], population) - temperature[i])
    return cost / len(time)
```

Décrivez les opérateurs mis en oeuvre

Mutation

1. Opérateurs qui échange les valeurs de b et c. En effet, b et c sont dans le même espace de recherche. C'est un échange de gène.

```
def b_c_flip_mutation(childrens, borne_b, borne_c, number_of_bc_flips = 5):
    index = np.random.choice(childrens.shape[0], size=number_of_bc_flips, replace=False)
    for i in index:
        childrens[i][1], childrens[i][2] = set_variable_value(childrens[i][2], borne_b, int), set_variable_value(childrens[i][1], bo
```

2. Opérateur qui redéfini totalement la valeur d'un poids au hasard dans son espace de recherche.

```
def reset_mutation(childrens, borne_a, borne_b, borne_c, number_of_resets = 10):
    rows = np.random.choice(childrens.shape[0], size = number_of_resets, replace=False)
    columns = np.random.choice(childrens.shape[1], size = number_of_resets)
    for i in range(number_of_resets):
        if columns[i] == 0:
            childrens[rows[i]][columns[i]] = set_variable_value(np.random.random(), borne_a, float)
        elif columns[i] == 1:
             childrens[rows[i]][columns[i]] = np.random.randint(borne_b[0], high=borne_b[1])
        else:
            childrens[rows[i]][columns[i]] = np.random.randint(borne_c[0], high=borne_c[1])
```

3. Opérateur qui modifie la valeur de a autour de sa valeur initiale d'après une loi normale centrée.

```
def a_mutation(childrens, borne_a, number_of_a_variations = 10, a_std_mutation = .2):
    variations = np.random.normal(scale=a_std_mutation, size=number_of_a_variations)
    np.random.shuffle(childrens)
    for i in range(number_of_a_variations):
        childrens[i][0] = set_variable_value(childrens[i][0] + variations[i], borne_a, float)
```

Croisement

L'opérateur de croisement prend 2 parents. Il créé deux enfants. Le premier enfant prend le b du parent 1 et le c du parent 2. L'enfant 2 le reste. Pour ce qui est de a, il est calculé une moyenne pondéré d'après une valeur α passé en paramètre.

$\forall\ \alpha \in]0, 1[$
$a_{enfant1} = \alpha \times a_1 (1 - \alpha) \times a_2$

$a_{enfant2} = \alpha \times a_2(1-\alpha) \times a_1$

```
def crossover(p1, p2, alpha_a_crossover = .3):
    c1, c2 = np.zeros(3), np.zeros(3)
    c1[1], c2[1] = p1[1], p2[1]
    c1[2], c2[2] = p2[2], p1[2]
    c1[0], c2[0] = p1[0] * alpha_a_crossover + p2[0] * (1 - alpha_a_crossover), p2[0] * alpha_a_crossover + p1[0] * (1 - alpha_a_cro
    return c1, c2
```

Décrivez votre processus de selection

La selection se passe en deux temps. La selection des parents et la selection des personnes qui vont être remplacé par les enfants. On admet pour la suite que lorsque notre fonction fitness est plus grande, l'individu est moins performant.

Pour la selection des parents, on va tirer sans remise des individu de façon inversement proportionelle à leur fitness.

```
def select_parents(population, fitness, number_of_parents = 20):
    prob = 1/fitness
    prob = prob / sum(prob)
    choices = np.random.choice(population.shape[0], size=number_of_parents*2, replace=False, p=prob)
    parents = []
    for i in range(0, len(choices), 2):
        parents.append([population[choices[i]], population[choices[i + 1]]])
    parents = np.array(parents)
    return parents
```

Pour la selection des individus qui vont être remplacés, on va tirer sans remise des individus parmis la population avec comme probabilité leur fitness. Pour conserver le meilleur individu, on lui associe une probabilité de 0.

```
def selection(population, childrens, fitness):
    tokeep = np.where(fitness == min(fitness))
    fitness = np.delete(fitness, tokeep)
    fitness = fitness/sum(fitness)
    change = np.random.choice(np.delete(np.arange(len(population)), tokeep), size=len(childrens), p=fitness, replace=False)
    for i in range(len(childrens)):
        population[change[i]] = childrens[i]
```

Quelle est la taille de votre population, combien de g´en´erations sont n´ecessaires avant deconverger vers une solution stable ?

Avec une taille de **population de 25**, Il faut environ **90 génération** avant de converger vers la meilleur solution. Cependant, les améliorations sont moindres à partir de 25 générations. Il y a donc de l'overfitting possible à partir de 25 générations.

Combien de temps votre programme prend en moyenne (sur plusieurs runs) ?

Avec des tests sur plusieurs centaines de génération, avec une population de taille 100, on trouve un moyenne de **10ms** et avec une population de taille 1000, on trouve une moyenne de **30ms**.

Discutez vos différentes solutions qui ont moins bien fonctionnées, décrivez-les et discutez-les

Il n'y a pas de solution qui ont moins bien fonctionnées. Tout a été question d'optimisation surtout sur la fonction de fitness qui prenait 95% du temps d'exécution. Pour l'optimiser, j'ai tout d'abord essayer de sauvegarder toutes les valeurs de cos possible pour le problème mais cela prenait trop de mémoire et la solution de `sparse_cosine_similarities` proposé par `numpy.cos` a été la plus rapide.

Aussi, le problème de cet exercie est que, quelque soit les valeurs donnée, l'algorithme va vouloir donner une valeur de c faible (2 ou 3) et moduler la valeur de a pour obtenir la meilleur fitness. En effet, en traçant le graphique, on voit bien que la courbe suit bien les points, malgrès le fait que la valeur de c ne soit pas la bonne.

Gestion du bruit

On approxime le bruit par une loi normale de paramètre (0, 0.1). J'ai ajouté du bruit lors du calcul de la température pour tenter de reproduire le bruit avec les paramètre estimé du bruit.

Une autre tentative a été de faire une nfft (FFT avec des points non-equidistant) mais étant donné la fréquence de certaines valeur et le peu de point, il est difficile d'enlever le bruit.

Une troisième tentative a été de créer un réseau de neuronne en créant des données bruité avec l'hypothèse de la loi normale et d'avoir en sortie de ce réseau des donéées débruités (sans succès...)

Choix des paramètres

Ma fonction fit comporte de nombreux paramètre. La taille de la population a été choisi arbitrairement par rapport à la taille de l'espace de recherche. Ensuite, j'ai obtenu les paramètre en comparant la vitesse de convergence selon les différentes valeurs. Les plus importants sont les paramètre concernant la mutation. Cela veut donc dire que l'opérateur de mutation est la méthode qui permet d'obtenir une convergence.