Универзитет у Новом Саду

Факултет техничких наука

ДОКУМЕНТАЦИЈА

Нелинеарно програмирање и еволутивни алгоритми

Пројекти задатак:

Snake Game

Студенти:

Наташа Лаковић SW2/19
Невена Прокић SW6/19

Опис проблема

У овом пројекту желимо да интерпетирамо аутоматско играње познате игрице “Snake game”. Помоћу тренинг сета података, научићемо наш програм да игра што паметније је могуће. Циљ игре је да се “змија”, низ покретних квадрата, константно креће према својој храни и сваки пут када је поједе, повећа се за један квадрат све док је такво кретање могуће.

~ Правила игрице:

1. Почетна “змија” је величине три квадрата
2. Величина хране је 1 квадрат
3. Храна се поставља рандом на било ком слободном пољу (слободна поља су она на којима се не налази “змија”)
4. Циљ кретања змије је да поједе храну
5. Сваки пут када поједе храну, змија се повећава за један квадрат
6. Дозвољени смерови кретања змије су лево, десно и напред
7. Игра се завршава ако змија удари у ивицу или у себе саму

Увод

Мотивација за рад на овом пројекту произилази из једне од најдражих игрица из детињства.

Проблему смо првенствено приступиле истаживањем различитих начина имплементације и израде решења која укључују генетски алгоритам. Готово сва решења овог проблема захтевају познавање неуронских мрежа те смо и том сегменту посветиле пажњу како бисмо разумеле суштину с обзиром да то није нама познато градиво.

Наше решење проблема укључује неуронску мрежу и генетски алгоритам који се користе приликом тренирања како би се у последњој генерацији видели најбољи резултати. Тренинг подаци се генеришу у току извршавања програма и тренирање се врши кроз 100 генерација . Свака генерација има 48 јединки, где свака јединка има 243 гена (због неуронске мреже).

Имплементација

Пројекат је подељен у два дела:

1. Логика програма
2. Визуелни приказ програма

Логика програма

~ Сажето:

У логици програма се налази хеуристика по којој се врши бодовање кретања сваке јединке. За сваку јединку бодује се да ли је умрла, колико је хране појела и понављање покрета. У односу на главу змије посматрају се суседне позиције и угао између главе змије и позиције хране. На основу тих параметара и хромозома одређује се наредни корак змије уз помоћ неуронске мреже која се такође налази у логици програма. За сваку генерацију тренирања генетски алгоритам формира следећу популацију, док је почетна популација генерисана рандом. Из сваке генерације се бирају најбољих 24 родитеља који ће украштањем офрмити 24 детета. Јединке које представњају децу имају малу вероватноћу да мутирају. У самом програму је имплементирана и рулетска селекција, међутим у нашем проблему се она није показала као најбоље решење, јединке ће брже напредовати уколико се бирају само најбоље међу њима.
*Неуронска мрежа је преузета.

~ Опис неких од функција:

Функција која бира најбоље родитеље из тренутне генерације да буду родитељи наредне генерације. На основу fitness-а односно који број бодова је јединка освојила се бирају најбољи родитељи.

*self._already_choose_parents је листа која се активира уколико је за избор родитеља укључена и рулетска селекција (да се не би понављали родитељи).

```python
def _best_parent_selection(self):
    self.parents = np.empty((self.number_of_parents, self.population.shape[1]))
    for i in range(int(self.number_of_parents)):
        get_max_fitness_index = np.where(self.fitness == np.max(self.fitness))[0][0]
        self.parents[i, :] = self.population[get_max_fitness_index, :]
        self._already_choose_parents.append(get_max_fitness_index)
        self.fitness[get_max_fitness_index] = -999999999
```

Функција рулетске селекције је имплементирана али није искоришћена. Од остатка популације (без 12 најбољих родитеља) сваки скор јединке се множи са рандом бројем између нула и један. Затим се бира најбољих 12 јединки тј. најбољих 12 скорова. Тиме се добија укупно 24 родитеља.

```python
def _roulette_selection(self):
    num_of_needed_parents = int(self.number_of_parents / 2)

    fitness_for_roulette = []
    for i in range(len(self.population)):
        if i not in self._already_choose_parents:
            fitness_for_roulette.insert(i, random.randint(0, 1) * self.fitness[i])
        else:
            fitness_for_roulette.insert(i, -9999999999)

    fitness_for_roulette = np.array(fitness_for_roulette)
    for i in range(num_of_needed_parents):
        get_max_fitness_index = np.where(fitness_for_roulette == np.max(fitness_for_roulette))[0][0]
        self.parents[i + 12, :] = self.population[get_max_fitness_index, :]
        self._already_choose_parents.append(get_max_fitness_index)
        fitness_for_roulette[get_max_fitness_index] = -999999999
```

Функција crossover имплементира стварање деце укрштањем родитеља. Број деце је једнак броју родитеља и сваки пар родитеља добија два детета. Бирају се рандом генерисани индекси родитеља и они се укрштају. Свако дете има 50 посто шанси да добије ген од првог родитеља и исто толико од другог.

```python
def crossover(self, number_of_crossover):  # number_of_crossover = (24,243)
    number_of_chromosome = number_of_crossover[0]
    number_of_gen = number_of_crossover[1]
    self.children = np.empty(number_of_crossover)

    for individual in range(int(number_of_chromosome / 2)):
        parent_index_random1 = random.randint(0, self.number_of_parents - 1)
        parent_index_random2 = random.randint(0, self.number_of_parents - 1)

        if parent_index_random1 != parent_index_random2:
            for gen in range(number_of_gen):
                if random.uniform(0, 1) < 0.5:
                    self.children[individual, gen] = self.parents[parent_index_random1, gen]
                else:
                    self.children[individual, gen] = self.parents[parent_index_random2, gen]
                if random.uniform(0, 1) < 0.5:
                    self.children[individual + 12, gen] = self.parents[parent_index_random1, gen]
                else:
                    self.children[individual + 12, gen] = self.parents[parent_index_random2, gen]
```

Функција која имплементира мутацију је приказана на слици испод. У њој се са малом вероватноћом бира јединка на којој ће се извршити мутација (промена на гену који је рандом изабран).

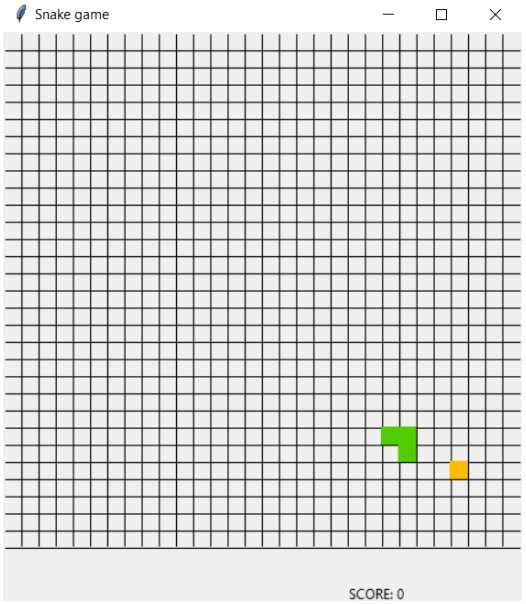
```python
    def mutation(self):
        number_of_children = len(self.children)
        number_of_gen = len(self.children[0])
        for individual in range(number_of_children):
            if random.uniform(0, 1) < 0.2:
                random_coeff_for_children_gen = random.randint(0, number_of_gen - 1)

                random_coeff = np.random.choice(np.arange(-1, 1, step=0.001), size=1, replace=False)
                self.children[individual, random_coeff_for_children_gen] =\
                    self.children[individual, random_coeff_for_children_gen] + random_coeff
```

Визуелни приказ

Визуелни приказ рађен је на два начина, али заједничке карактеристике су табла игре која представља матрицу 30x30, постављање хране на један од слободних квадрата и приказивање сваког померања змије.

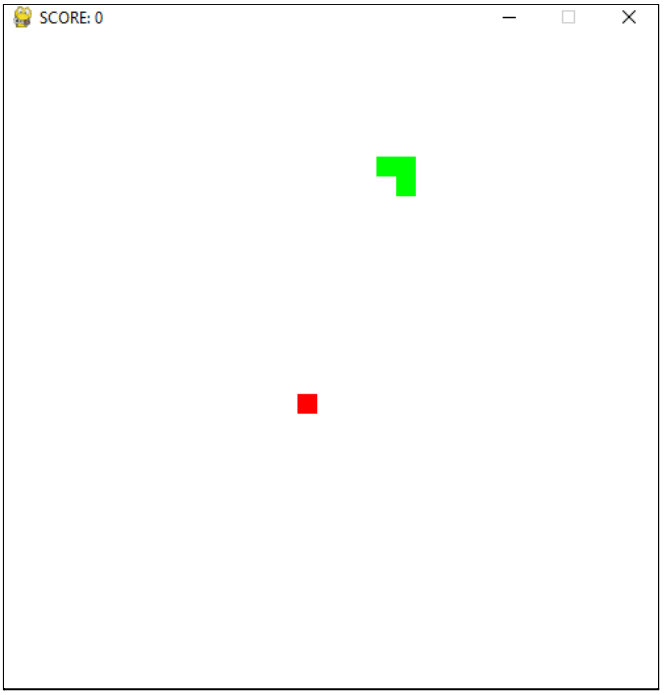
1. Реални приказ

Најважнија карактеристика овог приказа да се промене кретања змије приказују нормалном брзином коју је могуће испратити, међутим како тренирање модела траје веома дуго, овај начин приказивања није ефикасан. Коришчена је библиотека tkinter, табла се састоји од 30 хоризонталних и 30 вериткалних, сваки настали квадрат (уколико је слободан) је поље намењено за кретање змије и постављање хране.



2. Убрзани приказ

Овај приказ рађен је помоћу библиотеке pygame. Ова библиотека има функцију која омогућава брз приказ промена и веома добро прати развој популације. Уколико желимо само да пратимо резултате и развојни пут нашег модела, овај приказ је много бољи. Када је реч о брзини, у питању је период око 45 до 60 минута у случају оваквог приказивања.



Закључак

Уочени проблем приликом саме имплементације је јако споро извршавање програма због графичког приказа змијице и због великог броја података. Могуће побољшање овог програма лежи у бољем познавању тематике неуронских мрежа, имплементацији дубље хеуристике предвиђања наредног покрета.

Комбинација генетског алгоритма и нуронске мреже представља добар спој за решавање оптимизационих проблема. Такође поред самих алгоритама постоји много фактора које је потребно предвидети и обрадити. Конкретно у нашем проблему то је одабир наредног корака на који би требало да утичу не само суседна поља и положај хране него и последице предвиђеног корака односно разматрање 3-4 корака у напред. Таква хеуристика је много захтевнија од оне која је имплементирана у овом пројекту. Уколико би се имплементирао такав начин одабира корака, алгоритам би постао спорији у погледу извршавања, али би брже постигао добре резутлате.

Генерални закључак је да се сваком оптимизационом проблему мора приступити темељно и за квалитетне резултате потребно је време и дубоко проучавање сржи проблема. Потребно је прилагодити алгоритме и методе како би се добили најбољи резултати.