Математичка гимназија

Матурски рад

из предмета Програмирање и програмски језици

Игре и неуронске мреже - аутоматизација играчког искуства

Ученик:

Лука Ђурић, IVб

Ментор:

Милош Арсић

Београд, мај 2024.

Садржај:

1. Увод

У савременом добу, технолошки напредак непрестано отвара нова врата за истраживање и примену различитих техника у области вештачке интелигенције. Једна од најузбудљивијих област овог истраживања свакако су неуронске мреже. Неуронске мреже представљају модел инспирисан радом људског мозга и играју кључну улогу у решавању сложених проблема, од препознавања слика до обраде природног језика.

У оквиру овог рада, фокусираћемо се на примену неуронских мрежа у контексту учења рачунара како би постигао вештину играња видео игрица. Ова област истраживања постаје све значајнија, јер не само што омогућава рачунарима да развијају стратегије и прилагођавају се динамичким окружењима, већ такође отвара пут ка дубљем разумевању механизама учења код вештачке интелигенције. Кроз анализу конкретних примера и истраживање постојећих метода, ова тема ће истражити како неуронске мреже трансформишу приступ учењу компјутера у свету видео игрица.

Размотрићемо кључне концепте неуронских мрежа и њихову примену у домену игара, истражујући како ови модели могу ефикасно да уче и да се прилагођавају променљивим условима игре.

Flappy Bird, иако једноставна игра у свом концепту, представља изазов у развоју, захтевајући пажљиву интеграцију графичких елемената, обраде корисничког уноса и динамичког управљања игром.

Циљ ове имплементације није само стварање дупликата игре Flappy Bird, већ и примена неуронских мрежа за постизање аутономног управљања ликом у игри. Ова иновативна перспектива додаје сложеност пројекту, уводећи елементе дубоког учења како би се омогућило лику да доноси одлуке о летењу кроз препреке.

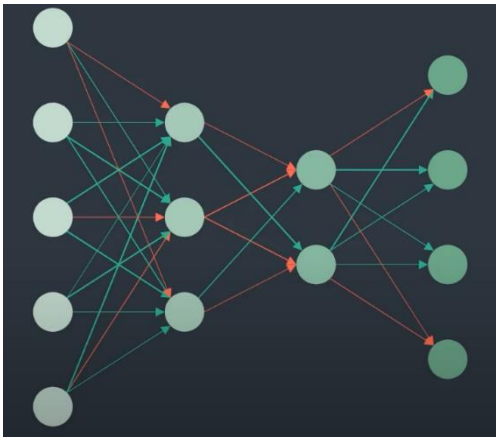
2. Неуронске мреже

Неуронске мреже су облик имплементације система вештачке интелигенције инспирисаних структуром и функционисањем људског мозга. Користе се за обављање различитих задатака попут препознавања узорака, класификације, прогнозирања и других облика обраде података.

2.1 Структура

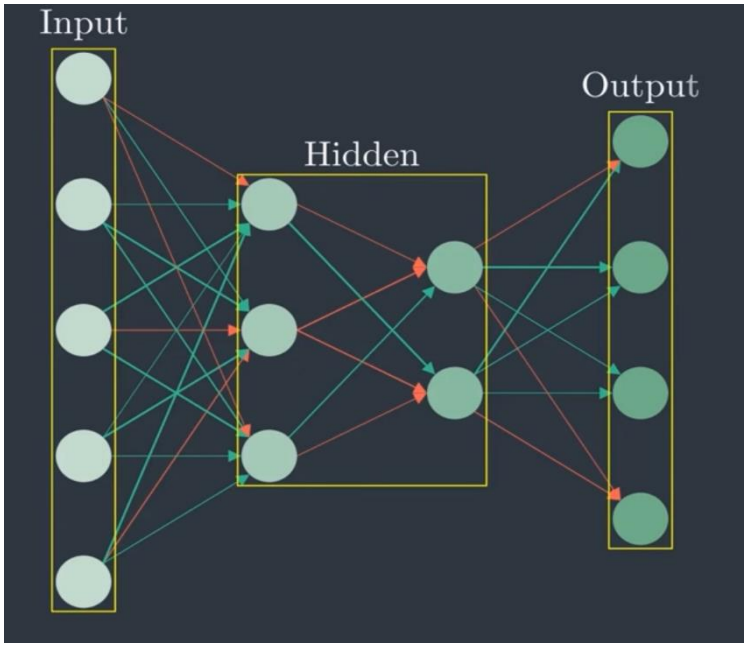
Основна јединица у неуронским мрежама је неурон, који је моделиран према стварним неуронима у мозгу. Неурон прима улазне податке, обрађује их и помоћу тежина (коефицијената) и функције активације генерише излаз. Неуронску мрежу можемо замислити као тежински усмерени граф код кога неурони представљају чворове, а синапсе представљају гране. Свака грана има одређену вредност која представља јачину повезаности између неурона.

Слика 1. Приказ неурона



Слика 2. Приказ неуронске мреже

Неуронске мреже састоје се од слојева неурона обично подељених у три главна типа: улазни слој (input), скривени слој (hidden) и излазни слој (output).



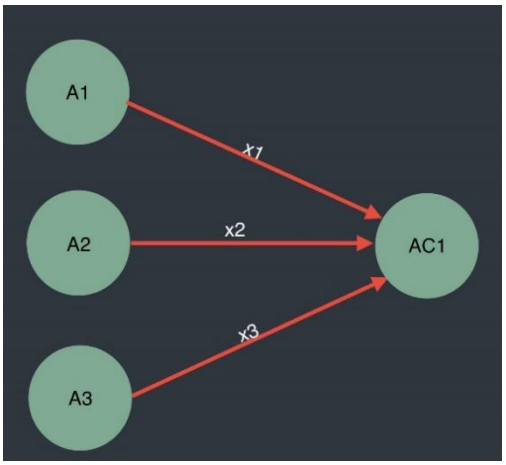
Слика 3. Неуронска мрежа са означеним слојевима

Улазни слој чине они неурони који представљају сирове вредности које се достављају неуронској мрежи. Скривени слој је сачињен од неурона који су корисни за трансформацију података док излазни неурони представљају коначне вредности.

2.2 Тежинска сума и Функција активације

Као што смо већ споменули неуронска мрежа може да се представи као тежински усмерени граф код кога неурони представљају чворове а синапсе представљају гране. Свака грана има одређену вредност која представља јачину повезаности између неурона (x1, x2, x3) док сваки чвор има своју тежину (A1, A2, A3).

Слика 4. Приказ неурона са тежином

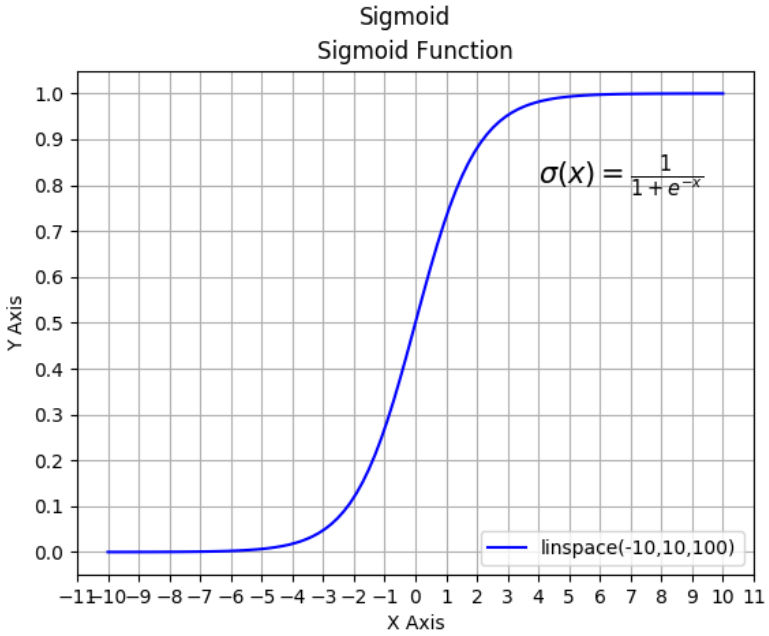


Слика 5. Приказ неуронске мреже са тежинама

Тежинска вредност неурона (AC1) представља суму свих тежина неурона помножену јачинама одговарајућих веза. Тежинска вредност дефинисана је формулом:

$$\boldsymbol{ACj = \sum AiXi}$$

Како тежинска вредност може бити било који број, потребан нам је неки праг како бисмо активирали неурон. Функција активације нам помаже да дефинишемо овај праг. Једна од често примењених функција активације је Сигмоидна функција која слика сваки број из интервала реалних бројева у број из интервала (0,1).



Слика 6. Сигмоид Функција

Међутим шта ако бисмо хтели да активирамо неурон за мање вредности? Ово ћемо постићи ако додамо **bias** тежинској суми пре него што применимо активациону функцију.

$$\boldsymbol{ACj = \sum AiXi + bias}$$

Овим померамо функцију у леву или десну страну у зависности од вредности bias-а, што нам омогућава да манипулишемо прагом активације.

2.3 Тренирање

Бирање тежина, bias-а і активационих функција се одређује тренирањем. Процес обуке неуронских мрежа укључује представљање мрежи скуп улазних података (енг. *training data*) и припадајућих очекиваних излаза. Током тренинга, мрежа

прилагођава тежине како би минимализовала разлику између стварних и очекиваних излаза.

Најпознатији алгоритам за тренирање неуронских мрежа је алгоритам повратне пропагације (енг. *backpropagation algorithm*) који користи технике појачаног учења (енг. *reinforcement learning*) где програм научи да направи одлуке помоћу интеракције са окружењем. Један од таквих алгоритама је и HEAT алгоритам. Пре него што се позабавимо HEAT алгоритмом потребно нам је разумевање начина на који генетски алгоритми функционишу.

2.4 Генетски алгоритми

Генетски алгоритми су дизајнирани тако да нађу решење проблема имитирајући основни принцип еволуције - преживљавање најприлагођенијег. Први корак је да се иницира популација. Свака јединка има сет гена названа геном или хромозом. Сваки геном представља могуће решење проблема које не мора нужно да буде добро, па нам је потребан начин да га проценимо. Евалуација је постигнута функцијом која узима геном и даје нам број који описује колико је ефикасан као решење за наш проблем. Зове се фитнес функција (енг. *fitness function*) и евалуациони број фитнес (енг. *fitness*).

Генетски алгоритми почињу са евалуацијом фитнес-а сваке индивидуе и бирају најбоље. Изабране индивидуе су изабране да представљају следећу генерацију. На пример можемо да изаберемо пар индивидуа и комбинујемо њихове геноме да направимо потомка слично као генетска комбинација у биологији. Главна идеја је мешање гена најбољих индивидуа са намером да можда добијемо још бољу јединку. Овај процес се назива кросинг - овер (енг. *crossing over*).

Понекад се случајне промене дешавају покушавајући да укрстимо гене индивидуа. Овакве промене се зову мутације у биологији и популацији омогућавају генетску разноврсност. Мутације су једини начин да врсте еволуирају тако да представљају фундамент нашег постојања. Алгоритам престаје када стигне до постигнутог фитнес-а или ако не може више да напредује.

3. Имплементација игре Flappy Bird

3.1 Увод

Flappy Bird је једноставна, али изазовна аркадна игра у којој играч контролише птицу која покушава да прође кроз узане просторе између вертикалних цеви. Играч контролише лет птице притиском на екран (или тастатуру), усмеравајући је кроз отворе у цевима. Циљ је постићи што већи број пролазака без додира са цевима или ивицом екрана.

Flappy Bird, направљен од стране вијетнамског програмера Донг Нгујена, првобитно је пуштен у јавност у мају 2013. године. Игра је стекла велику популарност због своје једноставности и дизајна. Међутим, због неочекиваног успеха, Донг Нгујен одлучио је да повуче игру из продаје у фебруару 2014. године, тврдећи да је успех игре постао терет за њега.

Након повлачења игре, Flappy Bird је стекао статус културног феномена, а многи програмери су добили инспирацију да стварају своје верзије игре. Ова игра оставља свој печат у историји видео игара, истовремено подсећајући на једноставност и изазов који може дефинисати популарност неке игре. Имплементација Flappy Bird игре са додатком неуронских мрежа представља покушај комбиновања класичног играчког искуства са модерним приступом дубоком учењу.

3.2 Преглед игре

За имплементацију игре користићемо објектно оријентисано програмирање у програмском језику Python. Можемо уочити 3 објекта: птица, позадина и цеви. Потребна нам је класа за сваки од датих објеката и такође бесконачна петља која ће се прекинути ако критеријуми за преживљавање птице нису испуњени.

3.3 Имплементација модула, потребних променљивих и слика

Модули које импортујемо су: `pygame`, `random`, `os`, `time`, `neat` и `pickle`. Иницијализујемо фонт помоћу `pygame.font.init()`.

Дефинишемо ширину и дужину екрана као и фонтове и величину слова

```
WIN_WIDTH = 600
WIN_HEIGHT = 800
FLOOR = 730
STAT_FONT = pygame.font.SysFont("comicsans", 50)
END_FONT = pygame.font.SysFont("comicsans", 70)
DRAW_LINES = False
```

Слике које ће се приказивати на екрану чувамо у листу:

```
bird_images=[pygame.transform.scale2x(pygame.image.load(os.path.join("slik
e","bird" + str(x) + ".png"))) for x in range(1,4)]
```

Иницијализујемо и слике базе, позадине и цеви:

```
pipe_img =
pygame.transform.scale2x(pygame.image.load(os.path.join("slike","pipe.png"
)).convert_alpha())

bg_img =
pygame.transform.scale(pygame.image.load(os.path.join("slike","bg.png")).c
onvert_alpha(), (600, 900))


base_img =
pygame.transform.scale2x(pygame.image.load(os.path.join("slike","base.png"
)).convert_alpha())
```

3.4 Класа Птица

На почетку дефинишемо потребне константе:

```
MAX_ROTATION = 25 # угао нагиба птице
IMGS = bird_images  # користимо листу слика
ROT_VEL = 20 # угаона брзина по фрејму
ANIMATION_TIME = 5 # време трајања анимације
```

Методе које имплементирамо су: **jump**, **move**, **draw** и **getmask**. Методу jump позивамо када желимо да се птица помери ка горе односно скочи.

```
def jump(self):
        self.vel = -10.5 # негативно јер x оса почиње од горе лево
        self.tick_count = 0 # наша јединица за време
        self.height = self.y
```

Методу **move** позивамо када птица треба да се помери:

```
def move(self):
        self.tick_count += 1

        displacement = self.vel*(self.tick_count) +
0.5*(3)*(self.tick_count)**2
        # d = vt + (1/2)at^2, пређени пут
        # ограничење пређеног пута да птица не би изашла са мапе
        if displacement >= 16:
            displacement = (displacement/abs(displacement)) * 16

        if displacement < 0: # подешавање да скокови изгледају природније
            displacement -= 2

        self.y = self.y + displacement

        # следећом if наредбом одређујемо коју слику да употребимо
        if displacement < 0 or self.y < self.height + 50: # нагиб на горе
            if self.tilt < self.MAX_ROTATION:
                self.tilt = self.MAX_ROTATION
        else:  # нагиб на доле
            if self.tilt > -90:
                self.tilt -= self.ROT_VEL
```

Метода **draw** нам је потребна да бисмо "анимирали" кретање птице:

```
def draw(self, win):

        self.img_count += 1

        # анимација птице
        if self.img_count <= self.ANIMATION_TIME:
            self.img = self.IMGS[0]
        elif self.img_count <= self.ANIMATION_TIME*2:
            self.img = self.IMGS[1]
        elif self.img_count <= self.ANIMATION_TIME*3:
            self.img = self.IMGS[2]
        elif self.img_count <= self.ANIMATION_TIME*4:
            self.img = self.IMGS[1]
        elif self.img_count == self.ANIMATION_TIME*4 + 1:
            self.img = self.IMGS[0]
            self.img_count = 0
```

```
        # када птица иде носем на доле не маше крилима
        if self.tilt <= -80:
            self.img = self.IMGS[1]
            self.img_count = self.ANIMATION_TIME*2


        # слика се ротира око (0,0)
        blitRotateCenter(win, self.img, (self.x, self.y), self.tilt)
```

И на самом крају метода **get_mask** нам враћа листу положаја пиксела птице чије употреба ће бити објашњена у наставку:

```
def get_mask(self):
        return pygame.mask.from_surface(self.img)
```

3.5 Класа Цев

Како се у имплементацији цеви крећу дуж x осе, потребна им је константна брзина која ће бити иста као и брзина позадине `VEL = 5`. "Пролаз" између цеви је такође константан `GAP = 200`.
Конструктор класе Цев:

```
def __init__(self, x):
      self.x = x
      self.height = 0

      # позиција горње и доње цеви
      self.top = 0
      self.bottom = 0

      self.PIPE_TOP = pygame.transform.flip(pipe_img, False, True)
      self.PIPE_BOTTOM = pipe_img
      self.passed = False # да ли је птица прошла цев

      self.set_height()
```

Методе које се имплементирају су: **set_height**, **move**, **draw** и **collide**.

Метода **set_height** нам враћа позицију горње и доње цеви на случајан начин док чува константно растојање.

```
def set_height(self):
```

```
        self.height = random.randrange(50, 450)
        self.top = self.height - self.PIPE_TOP.get_height()
        self.bottom = self.height + self.GAP
```

Метода **move** нам је потребна када желимо да се цев креће.

```
def move(self):
        self.x -= self.VEL
```

Метода **draw**, као што јој и само име говори, црта цеви на екран.

```
def draw(self, win):
        win.blit(self.PIPE_TOP, (self.x, self.top)) # горња
        win.blit(self.PIPE_BOTTOM, (self.x, self.bottom)) # доња
```

Последња и најбитније метода **collide** која нам даје информацију да ли се цев и птица додирују.

```
def collide(self, bird, win):
        bird_mask = bird.get_mask() #садржи позиције свих пиксела
        top_mask = pygame.mask.from_surface(self.PIPE_TOP)
        bottom_mask = pygame.mask.from_surface(self.PIPE_BOTTOM)

        top_offset = (self.x - bird.x, self.top - round(bird.y))
        bottom_offset = (self.x - bird.x, self.bottom - round(bird.y))

        b_point = bird_mask.overlap(bottom_mask, bottom_offset)
        t_point = bird_mask.overlap(top_mask,top_offset)

        if b_point or t_point: #ako se dodiruju
            return True
        return False
```

3.6 Класа База

Како се птица креће по y оси а цеви се крећу по x оси, база ће се кретати истом брзином и у истом смеру као цеви `VEL = 5`. Такође су нам потребни дужина `WIDTH = base_img.get_width()` и слика базе `IMG = base_img`.

Конструктор класе:

```
def __init__(self, y):
        self.y = y
        self.x1 = 0
```

```
        self.x2 = self.WIDTH
```

Класа База садржи две методе: **move** и **draw**.

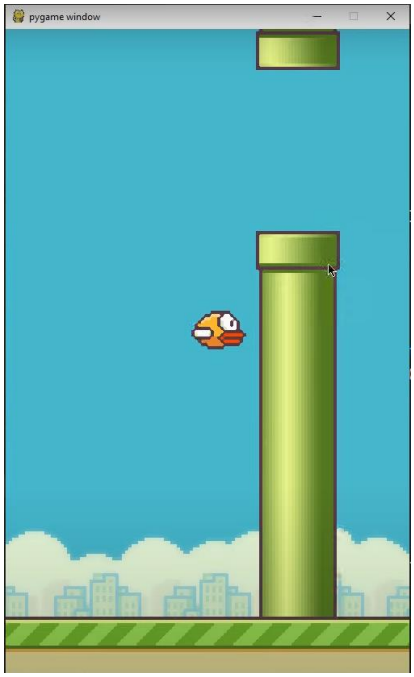
Метода **move** функционише тако што ми померамо базу тако да изгледа као да се "слајдује", цртамо две слике и померамо их док прва не изађе са екрана и затим је пребацујемо на почетну позицију.

```
def move(self):

        self.x1 -= self.VEL
        self.x2 -= self.VEL
        if self.x1 + self.WIDTH < 0:
            self.x1 = self.x2 + self.WIDTH

        if self.x2 + self.WIDTH < 0:
            self.x2 = self.x1 + self.WIDTH
```

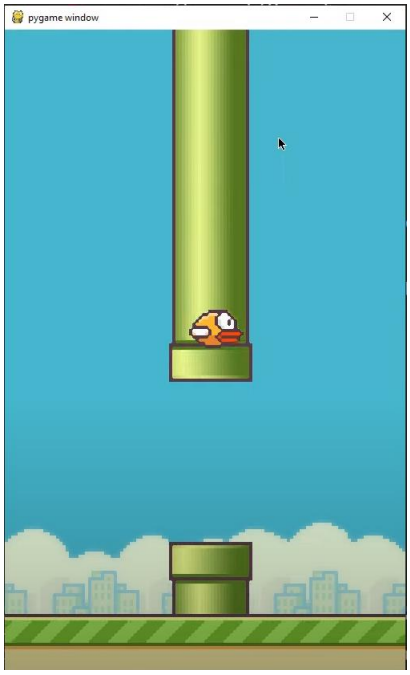
Метода **draw** црта у нашем случају две базе.

```
def draw(self, win):
        win.blit(self.IMG, (self.x1, self.y))
        win.blit(self.IMG, (self.x2, self.y))
```

Тренутно апликација изгледа овако:



Слика 7. Приказ апликације 1



Слика 8. Приказ апликације 2

4. HEAT Алгоритам

4.1 Увод

HEAT (енг. *Neuro Evolution of Augmenting Topologies*) је алгоритам за еволуцију неуронских мрежа. Овај алгоритам комбинује генетске алгоритме и машинско учење како би оптимизовао структуру и тежине неуронске мреже истовремено. Укратко, HEAT алгоритам ради на следећи начин:

1. **Иницијализација**: Започиње са популацијом једноставних неуронских мрежа које садрже само улазне и излазне неуроне.

2. **Евалуација**: Свака мрежа у популацији се евалуира на основу одређене функције вредновања (фитнес функције), која мери колико добро мрежа решава проблем.

3. **Селекција**: Мреже са бољим резултатима имају већу вероватноћу да буду изабране за репродукцију.

4. **Кросовер и мутација**: Изабране мреже се комбинују (кросовер) и понекад модификују (мутација) како би се створиле нове мреже. HEAT такође уводи нове неуроне и везе током овог процеса, чиме се топологија мрежа повећава и постаје сложенија током времена.

5. **Замена**: Новонастале мреже замењују део или целу популацију и процес се понавља.

Главне предности HEAT алгоритма су:

- **Прогресивна сложеност**: Мреже почињу као једноставне и постају сложеније током времена, што олакшава процес еволуције.

- **Специјализација**: Алгоритам чува иновације тако што одржава различите топологије у одвојеним групама, омогућавајући различитим решењима да се истовремено развијају.

- **Ефикасност**: Способност да истовремено оптимизује и топологију и тежине мреже чини HEAT ефикасним за сложене проблеме.

4.2 Модел мреже

Да бисмо омогућили рачунару да игра игрицу самостално потребно је да неуронској мрежи достављамо све неопходне информације.

Податке које дајемо мрежи су позиција птице и удаљеност птице од доње и горње цеви.

Податак који треба да добијемо је да ли птица треба да скочи.

У другом поглављу овог рада смо говорили о активационој функцији и користићемо функцију која је тамо наведена као пример (Сигмоидна функција).

Величина популације биће 50. Слични резултати би се добили који год број да ставимо.

Одабир фитнес функције нам је кључан моменат. Како ми желимо да укрштамо птице са најбољим учинком потребно нам је да их прво одредимо, што је циљ фитнес функције. Како је наша игрица једноставна и сама фитнес функција се лако одређује. За сваки пређени пиксел односно сваки померај на x оси наша птица добија „поен" што нам даје најквалитетније птице.

Максималан број генерација ћемо поставити на 30. То значи да ако после тридесете генерације немамо траженог најефикаснијег потомка софтвер се аутоматски искључује. У том случају нисмо имали среће.

4.3 Конфигурациони фајл

Конфигурациони фајл нам је потребан да бисмо рачунару рекли све из дела 4.2.

Прво програмирамо „NEAT" секцију. У следећем сегменту кода fitness_criterion нам означава да ли узимамо птице са највећим, средњим или минималним вредностима фитнес функције, у нашем случају ће то бити максимална вредност. Fitness_threshold нам означава вредност фитнеса после којег нема поенте наставити програм пошто ће птице преживети врло вероватно заувек. Pop_size нам представља величину популације и reset_on_extinction је сам по себи јасан.

```
[NEAT]

fitness_criterion = max

fitness_threshold = 100

pop_size = 50

reset_on_extinction = False
```

Следећа битна секција је „DefaultGenome” секција која нам дефинише почетне вредности генома, activation_default је Сигмоидна функција о којој смо причали раније, activation_mutate_rate представља шансу промене активационе функције што ће у нашем случају бити 0 јер нема потребе за компликовањем.

[DefaultGenome]

activation_default = tanh

activation_mutate_rate = 0.0

activation_options = tanh

Можемо да мењамо и вредности биас-а, с обзиром да не желимо да нам биас достиже вредности које би пореметиле популације ово су нам неке вредности које у пракси добро раде са овом апликацијом. Биас није константан зако нам је и потребна шанса мутација и вредност мутације као и минимална и максимална вредност биас-а

bias_init_mean = 0.0

bias_init_stdev = 1.0

bias_max_value = 30.0

bias_min_value = -30.0

bias_mutate_power = 0.5

bias_mutate_rate = 0.7

bias_replace_rate = 0.1

Један од битнијих делова секције је „DefaultGenome” чине следећа три параметра која говоре колико имамо података којих дајемо и колико оних које добијамо.

num_hidden = 0

num_inputs = 3

num_outputs = 1

Овде су објашњени најбитнији делови конфигурационог фајла док сви остали нису толико битни за наш софтвер. У случају да некога интересује више, сва објашњења осталих делова можете наћи на линку у секцији Литература [1].

4.4 Имплементација NEAT алгоритма

Стигли смо до последњег дела наше апликације, а то имплементација неуронске мреже и завршетак игрице. Прво ћемо да напишемо следећих пар линија које нам омогућавају да увек имамо приступ конфигурационом фајлу где год се он налазио.

```
if __name__ == '__main__':
    local_dir = os.path.dirname(__file__)
    config_path = os.path.join(local_dir, 'config-feedforward.txt')
```

Сада треба имплементирати алгоритам у наш програм:

```
def run(config_file):

    # укључујемо neat алгоритам да игра игрицу
      Config = neat.config.Config(neat.DefaultGenome,
neat.DefaultReproduction,neat.DefaultSpeciesSet,
neat.DefaultStagnation,config_file) # дефинишемо све под главе које имамо
у конфигурационом фајлу

    # креирамо популацију
    p = neat.Population(config)
    # вид праћења генерација
    # исписујемо статистику на терминалу по генерацији
    p.add_reporter(neat.StdOutReporter(True))
    stats = neat.StatisticsReporter()
    p.add_reporter(stats)
    winner = p.run(eval_genomes, 50)
    # приказујемо финалну статистику
    print('\nBest genome:\n{!s}'.format(winner))
```

Ово је све што нам је потребно да укључимо алгоритам у наш програм, међутим остаје нам да имплементирамо фитнес функцију која нам је кључна.
Најлогичније је да наша main функија буде заправо и фитнес функција.
Објаснићемо кључне делове main функције.

На следећи начин одређујемо да ли тренутно „посматрамо” прву или другу цев.

```
pipe_ind = 0
        if len(birds) > 0:
            if len(pipes) > 1 and birds[0].x > pipes[0].x +
      pipes[0].PIPE_TOP.get_width():
```

```
                pipe_ind = 1
```

Сваки пут када птица успешно пређе препреку ми „хранимо” њен фитнес:

```
        for x, bird in enumerate(birds):
            ge[x].fitness += 0.1
            bird.move()
```

Шаљемо локацију птице, горње и доње цеви и на основу одговора гледамо да ли да скачемо или не. Узимамо да када је output већи од 0.5 скачемо. Због сигмоидне функције вредности које добијамо су између -1 и 1.

```
      output = nets[birds.index(bird)].activate((bird.y, abs(bird.y -
pipes[pipe_ind].height), abs(bird.y - pipes[pipe_ind].bottom)))
      if output[0] > 0.5:
            bird.jump ()
```

Крећемо се кроз листу цеви и проверавамо да ли има поклапања између цеви и птице у случају да има смањујемо фитнес за 1. У случају да цев „изађе” са екрана бришемо је међутим треба обратити пажњу да не можемо да је избришемо одмах с обзиром да идемо кроз низ.

```
for pipe in pipes:
            pipe.move()
            for bird in birds:
                if pipe.collide(bird, win): # провера поклапања
                    ge[birds.index(bird)].fitness -= 1
                    nets.pop(birds.index(bird))
                    ge.pop(birds.index(bird))
                    birds.pop(birds.index(bird))

            if pipe.x + pipe.PIPE_TOP.get_width() < 0:
                rem.append(pipe)

            if not pipe.passed and pipe.x < bird.x:
                pipe.passed = True
                add_pipe = True
```

Када прођемо цев повећавамо резултат и такође додајемо нову цев на крај низа.

```
if add_pipe:
      score += 1
      for genome in ge:
            genome.fitness += 5
      pipes.append(Pipe(WIN_WIDTH)
```

Такође у случају да птица „изађе” ван ивица екрана треба је обрисати:

```
for bird in birds:
      if bird.y + bird.img.get_height() - 10 >= FLOOR or bird.y < -50:
nets.pop(birds.index(bird))
            ge.pop(birds.index(bird))
            birds.pop(birds.index(bird))
```

5. Закључак

Игра је једноставна тако да са великим бројем птица по генерацији шанса за генерисањем оне која ће бесконачно летети експоненцијално расте. Имплементација се може унапредити уношењем више улазних параметара неуронској мрежи (на пример: позиције левог и десног ћошка или горње и доње цеви). Такође комплекснији проблем би настао уколико би задали позицију само једне цеви, а да рачунар мора сам да одреди растојање између две цеви. Импленетација комплекснијег проблема је остављено за даљи рад на истраживању.

6. Литература

[1] Cesar Gomes Miguel, Carolina Feher da Silva, and Marcio Lobo Netto, *NEAT-Python's documentation,* https://neat-python.readthedocs.io: https://neatpython.readthedocs.io/en/latest/config_file.html датум последњег приступа:06.05.2024.

[2] Jp Patel, Sigmoid Function: *Derivative and Working Mechanism* 21 Mar 2024, /www.analyticsvidhya.com/: https://www.analyticsvidhya.com/blog/2022/12/sigmoid-function-derivative-and-working-mechanism/, датум последњег приступа: 06.05.2024.

[3] Prof. Risto Miikkulainen, *Neural Networks Research Group*, Department of Computer Sciences at the University of Texas at Austin: https://nn.cs.utexas.edu/, датум последњег приступа:07.05.2024.

[4] Kenneth O. Stanley and Risto Miikkulainen, *Efficient Evolution of Neural Network Topologies,* Department of Computer Sciences The University of Texas at Austin Austin, TX 78712, Proceedings of the 2002 Congress on Evolutionary Computation (CEC '02). Piscataway, NJ: IEEE: https://nn.cs.utexas.edu/downloads/papers/stanley.cec02.pdf датум последњег приступа:07.05.2024.