Neuroevolucijski algoritmi u igranju igara
Tehnička dokumentacija
Verzija 1.0

Studentski tim: Tvrtko Zadro

Nastavnik: prof. dr. sc. Marin Golub

Sadržaj

Tehnička dokumentacija

1. Opis razvijenog proizvoda

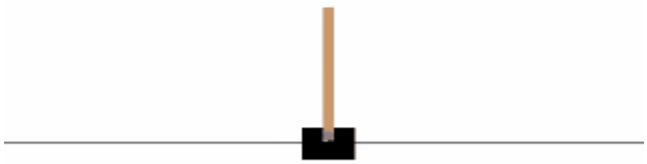
Ovaj projekt nastavlja se na diplomski seminar i programsko je ostvarenje opisanog neuroevolucijskog algoritma. Proizvod koristi gotov okvir koji omogućava igranje određenih igara iz programskog koda. Te igre koriste se kao platforma na kojoj se trenira neuroevolucijski algoritam. Cilj projekta je imati fleksibilno sučelje kojem se predaje opažanje igrača, a algoritam vraća potez igrača. Da bi se algoritam mogao trenirati, igra mora vratiti rezultat koji je algoritam postigao. U nastavku će biti opisan okvir igara koji podržava te opcije i algoritam za treniranje autonomnog igrača.

1.1 OpenAI Gym

Gym je naziv Python okvira kojeg je razvila firma OpenAI da bi testirali svoje algoritme potpornog učenja. S obzirom na to da tvrtka razvija svoje algoritme umjetne inteligencije, kao najbolji prvi korak čini se testirati te algoritme na igranju igara. To je jeftin i u novije vrijeme sve pouzdaniji način testiranja novih algoritama umjetne inteligencije uzevši u obzir kako se kompleksnost igara povećava iz godine u godinu.

Ovo sučelje omogućava korisniku da unutar Python programskog koda odabere jednu od igara i natrag dobije objekt pomoću kojeg vrši opservacije unutar igre, vuče poteze i na kraju, gleda rezultat odigrane igre.

U ovom projektu algoritam je testiran na igri CartPole-v0. Ovu igru koristi se kao primjer testiranja neuroevolucijskog algoritma promjenjivih topologija u originalnom radu [1]. Radi se o igri koja simulira balansiranje štapa na pomičnim kolicima. Svaki odigrani potez igrač dobije 1 bod, a igra završava kada štap padne ispod 15 stupnjeva od vertikalnog položaja. Cilj igre je sakupiti što više bodova. U svakom trenutku igre, sustav vraća opažanje u obliku četiri broja: pozicija kolica, brzina kolica, pozicija štapa i kutna brzina štapa. To opažanje algoritam koristi kako bi odredio hoće li u tom potezu kolica pomaknuti lijevo ili desno (Slika 1.1). U ovom slučaju, algoritam koji to radi je neuroevolucija promjenjivih topologija.



Slika 1.1 Slika iz igre *CartPole-v0*

1.2 Neuroevolucija promjenjivih topologija

Proizvod koji je tema ovog projekta implementira algoritam neuroevolucije promjenjivih topologija u programskom jeziku Python. Kao referenca za ostvarenje korišten je originalni rad kojeg je Ken Stanley predložio 2002. godine [1]. Ukratko, algoritam nadograđuje na ideju neuroevolucije tako što ne razvija samo težine veza neuronske mreže, nego i samu strukturu mreže. Inače se u algoritmima neuroevolucije struktura mreže definira unaprijed, a svaka jedinka unutar gena nosi informacije o težinama te strukture. U ovom slučaju, mreža počinje jednostavno, kao potpuno povezana mreža ulaznog i izlaznog sloja, a kroz generacije ta struktura se mutacijama može zakomplicirati. Budući da je u pozadini genetski algoritam, ideja je da mreža razvija samo elemente strukture koji osiguravaju poboljšanje učinkovitosti algoritma. U sljedećem poglavnju bit će opisani detalji algoritma i metode kojima je ostvaren ovaj algoritam.

2. Tehničke značajke

Budući da okvir za igranje igara nije predmet ovog projekta, njegov rad neće biti opisan, nego će biti opisane značajke neuroevolucije promjenjivih topologija. Programsko ostvarenje ovog algoritma dijeli se na dva glavna dijela: konstrukcija i pokretanje neuronskih mreža koje su opisane svakom jedinkom populacije i genetski algoritam koji manipulira populacijom.

2.1 Konstrukcija neuronskih mreža

Za početak, bitno je naglasiti da je opis svake neuronske mreže spremljen u jedinku kao niz veza između čvorova neuronske mreže. Zadatak ovog dijela je iz tog genotipa konstruirati fenotip, koji je onda spreman za korištenje. Fenotip se koristi za evaluaciju danih opažanja i na svom izlazu daje odlučeni potez. Dakle, svaka jedinka u sebi ima opis neuronske mreže koja predstavlja logiku odlučivanja jedinke.

Tijekom faze evaluacije jedinki rješenja, svaka jedinka konstruira svoj fenotip pomoću klase *Phenotype*. Klasao kao inicijalizacijski argument prima genotip jedinke, tj. sve veze unutar neuronske mreže koju jedinka predstavlja. Klasa onda iterira po svim vezama i stvara objekte tipa *Neuron* u kojima onda sprema dolazne, izlane veze i broj neurona koji moraju okinuti da bi neuron okinuo. Ovaj broj nije jednak broju ulaznih veza, jer su u algoritmu omogućene povratne veze. Na ovaj način neuron kada okine uzme trenutnu vrijednost neurona koji su povratna veza što zapravo odgovara vrijednosti iz prethodne iteracije.

Objekt klase se nakon toga koristi za prolazak mreže unaprijed i funkcionira tako da se postave vrijednosti ulaznih neurona. Svaki neuron nakon što mu se postavi vrijednost emitira to izlaznim neuronima. Izlazni neuroni onda ažuriraju broj neurona koji je potreban da bi trenutni neuron izračunao svoju vrijednost. Ako taj broj padne na nulu, neuron uzme ulazne veze i računa svoju vrijednost. Kao prijenosna funkcija koristi se sigmoida.

2.2 Genetski algoritam

Genetski algoritam zadužen je za praćenje efikasnosti jedinki i evaluaciju. Na početku procesa učenja, algoritam inicijalizira populaciju zadane veličine. Jedinke su inicijalizirane nasumično. Glavna petlja algoritma evaluira jedinke, sortira jedinke u vrste te generira novu generaciju (Slika 2.1).

```
population = Population()
for i in range(config.num_iter):
    print('Generation: {:d}, best fitness: {:.2f}'.format(i, population.evaluate_fitness()))
    population.speciate()
    population.breed_new_generation()
```

Slika 2.1 Glavna petlja genetskog algoritma

2.2.1 Evaluacija

Prilikom evaluacije populacije, svaka jedinka konstruira svoj fenotip i igra jednu partiju igre. Nakon što igra završi, rezultat koji je jedinka postigla koristi se kao dobrota.

2.2.2 Specijacija

Jedan od glavnih principa neuroevolucijskih algoritama promjenjivih topologija, u daljnjem tekstu NEAT (eng. *NeuroEvolution of Augmenting Topologies*), je specijacija. Taj proces dijeli sve jedinke populacije u pojedine vrste. Razlog zašto je ovo bitno je taj što je moguće veliko odstupanje u jedinkama. I neke jedinke koje razviju neki potencijalno dobri konstrukcijski element u početku neće imati dobar performans, zbog čega će brzo ispasti iz populacije. Specijacijom postižemo da se jedinke natječu unutar sličnog okruženja.

Specijacija se odvija tako da se za svaku jedinku računa udaljenost od predstavnika postojećih vrsti. Jedinka se pridodijeli prvoj vrsti kojoj joj je udaljenost od predstavnika manja od praga kompatibilnosti. Ako jedinka ne zadovoljava uvjet nijedne vrste, stvara se nova vrsta i jedinka postaje predstavnik. Formula računanja udaljenosti prikazana je na Slici 2.2. *E* (eng. excess) predstavlja broj gena koje jedan od jedinki nema, a inovacijski brojevi tih gena su veći od najvećeg inovacijskog broja te jedinke. *D* (eng. *disjoint*) predstavlja broj svih ostalih gena koji nema jedna od jedinki, a zadnji element predstavlja prosječnu vrijednost razlike između težina gena s jednakim inovacijskim brojem.

```
delta = (config.c1 * E) / N + (config.c2 * D) / N + config.c3 * sum(weight_diffs) / len(weight_diffs)
return delta
```

Slika 2.2 račun udaljenosti između dvije jedinke

2.2.3 Križanje

Važno je napomenuti da svaki gen unutar genoma ima svoj inovacijski broj koji mu se dodjeli kad se generira nova veza. Na ovaj način, kod križanja, točno se može znati koje gene uspoređivati s kojima. Križanje se radi tako da se uspoređuju svi parovi gena roditelja. Ako oba roditelja imaju gen s tim inovacijskim brojem, djetetu se dodjeljuje nasumično jedan, a ako jedan od roditelja nema gen s tim inovacijskim brojem djetetu se dodjeljuje postojeći.

Kod stvaranja nove generacije, svakoj vrsti se, ovisno o zbroju dobroti njenih jedinki, dodjeljuje broj nove djece. Nakon toga nasumično se izabiru jedinke iz vrste koje se križaju. Nove jedinke se mutiraju i dodaju u novu generaciju. Važno je napomenuti da nove jedinke ne pripadaju odmah istoj vrsti, već se vrsta svih jedinki ponovo računa u sljedećoj generaciji.

2.2.4 Mutacija

Postoje tri vrste mutacije. Mutacija veza, generiranje nove veze i generiranje novog čvora. Tijekom mutacije jedinke moguće je da se dogode sve tri mutacije (Slika 2.3).

```
def mutate(self, generation_innovations):
    if random() < config.connection_mutation_probability:
        self.mutate_connections()

    if random() < config.new_connection_probability:
        self.new_connection(generation_innovations)

    if random() < config.new_node_probability:
        self.new_node()
```

Slika 2.3 mutacija jedinke

Mutacija veze iterira po svim genima i uz neku vjerojatnost mutacije mijenja vrijednost težine veze. Jedna moguća opcija je da se težini doda vrijednost iz gaussove distribucije kojoj je μ trenutna vrijednost težine, a druga opcija je da se težini dodijeli nasumična vrijednost (Slika 2.4).

```
def mutate_connections(self):
    for connection in self.connections.values():
        if random() < config.perturbation_probability:
            connection.weight = connection.weight + gauss(config.step_mu,
                                                          config.step_sigma)
        else:
            connection.weight = random() * 2 - 1
```

Slika 2.4 mutacija veza

Generiranje novih veza je ostvareno tako da se biraju dva čvora koja nisu međusobno povezana. Jedinka čuva sortirane parove veza radi brze pretrage. Budući da su povratne veze moguće, ne provjerava se je li usmjerenje nove veze strogo u unaprijednom smjeru. Nova veza dobiva sljedeći dostupan inovacijski broj i veza se sprema u listu generacijskih inovacija. Ovo je potrebno kako se ne bi dogodilo da neka druga jedinka u istoj generaciji stvori istu vezu, ali na drugoj poziciji. Na ovaj način izbjegavamo duple veze kod križanja jedinki.

Stvaranje novih čvorova radi se tako da se izabere postojeća veza, onemogući se i stvori se novi čvor s koji će biti nova poveznica prethodna dva čvora. Ovo je jednostavnije ostvariti jer se ne mora ništa provjeravati.

3. Upute za korištenje

Rješenje se koristi tako da se unutar glavne Python datoteke definiraju hiperparametri algoritma i igra koja se igra (Slika 3.1). Nakon toga, dovoljno je u konzolu upisati komandu za pokretanje skripte (Slika 3.2). Nakon pokretanja otvorit će se prozor s igrom u sklopu *gym* okvira (Slika 3.3), a algoritam će u konzolu ispisivati detalje o izvođenju treniranja algoritma.

```
class Config:
    def __init__(self):
        self.connection_mutation_probability = 0.8
        self.perturbation_probability = 0.9
        self.new_node_probability = 0.04
        self.new_connection_probability = 0.08
        self.step = 0.25
        self.innovation_number = 8
        self.node_key = 6
        self.c1 = 1.0
        self.c2 = 1.0
        self.c3 = 0.8
        self.compatibility_threshold = 1.0
        self.crossover_probability = 0.75
        self.disable_probability = 0.75
        self.input_keys = [0, 1, 2, 3]
        self.output_keys = [4, 5]
        self.actions = {
            self.output_keys[0]: 0,
            self.output_keys[1]: 1
        }
        self.pop_size = 30
        self.num_iter = 70
        self.num_to_remove = 2
        self.new_mu = 0
        self.new_sigma = 1
        self.step_mu = 0
        self.step_sigma = 1
```

Slika 3.1 hiperparametri algoritma

```
C:\WINDOWS\system32\cmd.exe

C:\Users\tzadr\Documents\Python\dipl-proj>python NEAT.py
```

Slika 3.2 naredba za pokretanje algoritma

Slika 3.3 izgled prozora u sklopu *gym* okvira

4. Literatura

[1] Miikkulainen, R., Stanley, K.O., Evolving neural networks through augmenting topologies. *http://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf*, 2002.