

Reinforcement Learning in Pacman aan de hand van Proximal Policy Optimization

Elias Nijs, Bavo Verstraeten

Artificiele Intelligentie
Vakgroep Toegepaste Wiskunde, Informatica en Statistiek
Universiteit Gent

23 December 2022

Reinforcement Learning in Pacman aan de hand van Proximal Policy Optimization

Elias Nijs, Bavo Verstraeten

Artificiele Intelligentie
Vakgroep Toegepaste Wiskunde, Informatica en Statistiek
Universiteit Gent

1. Introductie

Pac-Man is een welbekend spel. Het is een spel met simpele regels, maar vergt toch enig strategisch inzicht. Dit is waarom Pac-Man volgens ons het perfecte spel is om te automatiseren.

1.1. Pacman

Eerst en vooral zullen we even de werking van het spel toelichten.

De speler bestuurt een wezen genaamd Pac-Man. De speler kan rondlopen in een groot doolhof. Hierin liggen er pellets op paden, en lopen ook spoken rond. De speler heeft als doel de spoken te ontwijken en alle pellets te verzamelen. Daarbovenop zijn er ook nog speciale pellets: de power pellets. Als Pac-Man zo een pellet verzamelt, gaat hij enkele seconden in een powerup state. Indien hij in deze state een spook aanraakt, sterft hij niet, maar verdwijnt deze geest enkele seconden van de map.

1.2. Agents

Het spel kent verschillende entiteiten die keuzes moeten maken. Pac-Man moet de paden afaan die zijn score maximaliseert en zijn kans op sterfen minimaliseert.

Daarnaast moeten ook de spoken keuzes maken. Zo zullen ze idealiter afslagen nemen die de kans om Pac-Man te verslaan vergroten.

Het leek ons interessant deze entiteiten tegelijk te trainen. Aangezien dit een spel is en er geen trainingsdata op voorhand beschikbaar is, zullen we gebruik maken van Reinforcement Learning. Op deze manier zouden beide entiteiten tegelijk proberen leren de andere entiteit te verslaan.

1.3. Onderzoeksvraag

Hieruit volgen onze oorspronkelijke onderzoeksvragen:

- Wint Pac-Man of winnen de geesten?
- Welke strategieën ontdekken zowel de Pac-Man agent als de geest agents?
- Welke invloed hebben het aantal geesten en de map lay-out op de vorige onderzoeksvragen?

Echter, na verloop van dit onderzoek, bleek al snel dat het onderzoeken van deze vragen heel wat tijd in beslag zou nemen. Diep in ons onderzoek zijn we dan tot de conclusie gekomen dat we deze tijd niet hadden. Dit heeft ons gedwongen onze onderzoeksvragen te herschalen.

We hebben beredeneerd dat het wel mogelijk is deze onderzoeksvragen te onderzoeken als we ons enkel focussen op het trainen van Pac-Man. Het trainen van de spoken is de factor die de schaal van dit project te groot maakt voor de tijd die we hadden.

Het nieuwe plan is dus om enkel de Pac-Man een eigen agent te geven met nog steeds als doel zoveel mogelijk pellets te eten en spoken te ontwijken, en de spoken willekeurig te laten rondbewegen. Dit zorgt voor gelijkaardige, maar andere, onderzoeksvragen:

- Kan Pac-Man winnen?
- Welke strategieën ontdekt Pac-Man?
- Welke invloed hebben het aantal spoken en de map lay-out op de vorige onderzoeksvragen?

1.4. Oplossingsmethoden

Er bestaan verschillende Reinforcement Learning Algoritmes, zoals DQN¹ en TRPO².

Na verschillende bronnen te raadplegen, zijn we

tot de conclusie gekomen dat PPO³ het stabielste en best presterende is van deze algoritmes. We zullen dan ook dit algoritme gebruiken om onze Pac-Man agent te trainen.

Een moeilijk deel van dit onderzoek was het bepalen van de hyperparameters om aan dit algoritme mee te geven. We hebben dan ook gebruik gemaakt van een library die uitgebreide algoritmes gebruikt om parameters van een functie te tunen.

Deze methoden worden uitgebreid besproken in het vervolg van dit verslag.

2. Methodiek

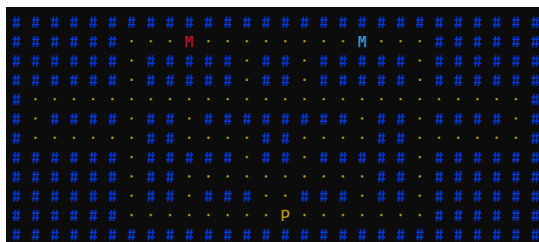
We bespreken nu uitvoerig hoe we dit onderzoek hebben uitgevoerd. In grote lijnen kunnen we dit opdelen in drie grote fasen:

1. Implementatie van het spel
2. Gebruik van Proximal Policy Optimization
3. Hyperparamter optimalisatie aan de hand van Optuna⁴

We bespreken elk van deze in detail.

2.1. Implementatie van Pac-Man

Het Pac-Man spel werd geïmplementeerd in python aan de hand van de ncurses bibliotheek. De ncurses bibliotheek levert verschillende manieren om een terminal-gebaseerde user interface uit te bouwen. Er werd gekozen om het spel zelf te maken om de implementatie zo simpel en uitbreidbaar mogelijk te houden.



In het begin van het programma wordt de ncurses bibliotheek geïmporteerd en wordt deze gebruikt om een nieuw venster binnen de terminal te creëren alsook om de controller voor de gebruikers input op te zetten.

Vervolgens wordt het spel geïnitieerd. Het spel wordt voorgesteld door een object dat een 2d array bevat. Deze array houdt de muren, pellets, ... bij. Verder houdt dit object ook Pac-Man en de

spoken bij, alsook enkele statistieken over de huidige staat van het spel.

Eens het spel geïnitieerd is, wordt de spel-lus opgestart. Deze bestaat uit enkele delen die telkens na elkaar worden opgeroepen. Deze zijn de volgende:

1. `kb_getqueue(screen)` om alle input op te halen.
2. `handleinput(game, screen)` om de ingelezen input af te handelen.
3. `gameupdate(game)` om op basis van de huidige staat en de input een nieuwe staat te creëren.
4. `gamerender(game)` om de nieuwe staat weer te geven.
5. Tot slot enkele lijnen om de frame-rate te begrenzen zodat wat er gebeurt interpreteerbaar blijft voor de gebruiker.

Wanneer we de agent zullen trainen worden stappen een, twee, vier en vijf weggelaten. Deze zijn enkel nodig indien een mens met het spel wilt interageren. Indien we deze niet weglaten, zouden we het trainingsproces significant en onnodig verlengen.

Tot slot werd om zowel het trainen te verbeteren als om het testen van de agent in een nieuwe omgeving mogelijk te maken, de functionaliteit om verschillende omgevingen in te laden gecreëerd. Voor het spel opstart, moet een omgeving geselecteerd worden. Tijdens de initialisatie zal het spel dan de geselecteerde omgeving inladen en parsen. Omgevingen worden gedefinieerd in een tekst bestand en doorgegeven via de locatie van dit bestand. Op deze manier wordt het zeer simpel om een nieuwe omgeving te maken en gebruiken.

Zo een omgeving ziet er als volgt uit:

```
#####
#P.....#
#.####.####.#
#.####.####.#
#.....#
#.#####.##.#
#.....G#
#####
```

Dit is de inhoud van het bestand `code/pac-man/maps/lv2.txt`.

We hanteren hierbij de volgende conventie.

#	<i>muur</i>
.	<i>pellet</i>
0	<i>power pellet</i>
	<i>gang</i>
P	<i>start positie Pac-Man</i>
G	<i>start positie spook</i>

De implementatie van het spel kan teruggevonden worden in de volgende bestanden:

1. functie implementaties:
code/pacman/pacman.py
2. datastructuren en constanten:
code/pacman/pacman_h.py

2.2. Proximal Policy Optimization (PPO)

PPO³ is een reinforcement algoritme dat aan populariteit gewonnen heeft in de laatste jaren. In dit onderzoek kijken we hoe dit algoritme zich gedraagt op een relatief simpele maar dynamisch omgeving aan de hand van het spel Pac-Man.

We bekijken eerst hoe PPO werkt en vervolgens hoe we dit algoritme gebruikt hebben.

2.2.1. Achtergrond PPO

PPO is een reinforcement learning algoritme ontwikkeld aan Open AI. Het algoritme heeft success in een breed veld aan taken, gaande van ataris spellen tot robot controllers tot een meer ingewikkeld spel zoals dota2.

Het grote probleem met reinforcement learning algoritmes is het feit dat de training data afhankelijk is van het huidige policy. Dit komt omdat de agent zijn eigen training data genereert door met de omgeving te interageren. Dit in tegenstelling tot supervised learning, waar we een statische dataset hebben om op te trainen.

Dit betekent dus dat de datadistributies over zowel de observaties als de rewards constant in flux zijn terwijl de agent aan het leren is. Terwijl we trainen ondervinden we dan ook een grote hoeveelheid onstabiliteit. Daarboven op zijn reinforcement learning algoritmes ook nog eens zeer gevoelig aan de hyperparameters en de waarden bij initialisatie, wat de stabiliteit van de algoritmes nog zal verlagen.

Het probleem met reinforcement learning algoritmes ligt er dus in om zo stabiel mogelijk te zijn, zonder computationeel te veel rekenkracht nodig

te hebben en tegelijkertijd het aantal nodige samples laag te houden. Daarnaast worden dit soort algoritmes ook vaak heel complex omwille van deze problemen. Een bijkomende moeilijkheid is dus om de complexiteit van de implementatie laag te houden.

PPO probeert deze moeilijkheden aan te pakken. De achterliggende doelen van het algoritme zijn de volgende

1. gemakkelijk implementeerbaar
2. sample efficient
3. makkelijk te tunen

PPO is een on-policy-gradient methode. Dit betekent dat het, in tegenstelling tot bijvoorbeeld DQN¹ die leert van opgeslagen offline data, PPO online leert. In plaats van een buffer met vorige ervaringen bij te houden zal PPO dus direct leren van een ervaring. Eens een batch ervaring gebruikt is, wordt deze weggegooid. Een nadeel hiervan is wel dat dit soort methoden minder sample-efficient zijn dan q-learning methodes.

We kijken eerst naar vanilla policy gradient methods:

$$L^{PG}(\theta) = \hat{E}_t \left[\log \pi_{\theta}(a_t | s_t) \hat{A}_t \right]$$

Het grote probleem hiermee is dat als we gradient-descent op een enkele batch van ervaringen blijven toepassen, dan zullen de parameters in het netwerk heel ver buiten het bereik van waar deze data verzameld was komen te liggen. Dit zorgt er bijvoorbeeld voor dat de advantage function, die een benadering van het echte advantage is, compleet fout wordt. We zijn op deze manier dus in principe ons policy aan het ondermijnen. Uit dit probleem komen we tot de volgende vraag:

Hoe kunnen we de stap met de grootste verbetering nemen op een policy met de data die we momenteel hebben, zonder dat we zo ver stappen dat de performantie ineens stort?

Om dit op te lossen kunnen we een beperking invoeren die beperkt hoe ver het nieuwe policy van het oude policy kan komen te liggen. Dit idee werd geïntroduceerd in de paper *Trust Region Policy Optimization (TRPO)*². Deze paper vormt ook de basis voor PPO.

De objective function in TRPO is de volgende:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

$$\underset{\theta}{\text{maximize}} \hat{E}_t[r_t(\theta)\hat{A}_t]$$

We zien een verandering van de log operatie in vanilla policy gradients naar een deling door $\pi_{\theta_{old}}$. Om ervoor te zorgen dat het nieuwe policy conservatief blijft, wordt hier nog een KL (Kullback-Leibler) ⁵ beperking aan toegevoegd:

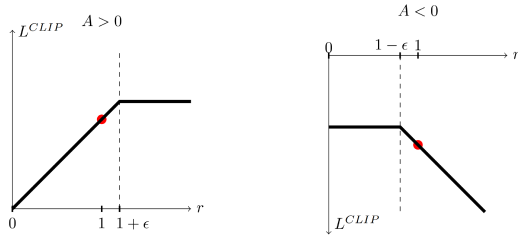
$$\hat{E}_t \left[KL[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)] \right] \leq \delta$$

Deze KL beperking leidt echter wel tot significante overhead tijdens de training. Hetgeen PPO hierop probeert te verbeteren is om deze beperking direct in ons optimalisatie-doel te verwerken.

We komen nu tot de objective functie van PPO.

$$\hat{E}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t \right) \right]$$

Merk op dat het probabiliteits ratio r geknipt wordt op $1 - \epsilon$ of $1 + \epsilon$ afhankelijk van het teken van \hat{A}_t . Dit is duidelijker te zien op de volgende afbeelding uit de originele paper:



Met PPO bereiken we dus hetzelfde doel als met TRPO zonder de grote overhead die meekomt met het bereken van de KL-divergentie. verder nog, uit metingen blijkt dat PPO vaak beter presteert dan TRPO.

We bekomen het volgende algoritme:

```

for  $i \leftarrow 1, 2, \dots$  do
  for  $actor \leftarrow 1, 2, \dots, N$  do
    run policy  $\pi_{\theta_{old}}$  for  $T$  steps
    compute  $\hat{A}_1, \dots, \hat{A}_T$ 
  done
  optimize surrogate  $L$  wrt  $\theta$ ,
  with  $K$  epochs
  and minibatch size  $M \leq NT$ 
   $\theta_{old} \leftarrow \theta$ 
done

```

Nu we PPO in detail bekeken hebben, kunnen we overgaan naar hoe we dit algoritme gebruikt hebben.

2.2.2. Gebruik PPO

2.2.2.1. Stable Baselines 3 ⁶

Het is natuurlijk niet de bedoeling dat we zelf deze ingewikkelde algoritmes implementeren. We gaan daarom gebruik maken van een library die voor ons deze algoritmes ter beschikking stelt. Het zelf implementeren zou te veel tijd kosten, en zou waarschijnlijk slechter presteren dan de implementaties van een library.

Na verschillende bronnen te raadplegen, hebben we besloten gebruik te maken van de library Stable Baselines 3 ⁶. Deze library biedt ons een heel eenvoudige manier van modellen configureren en trainen. Een algoritme genaamd 'ALGO', trainen met hyperparameters 'HYPER' voor 'STEPS' stappen gebeurt door

```
ALGO(Policy, Env, HYPER).learn(STEPS)
```

op te roepen. De enige parameters die we nog niet in dit verslag besproken hebben, zijn Policy en Env.

De Policy bepaalt hoe ons neurale netwerk zelf er uit ziet, en hoe zijn nodes gestructureerd zijn. Wij hebben gekozen voor een Multilayer Perceptron Policy, wat de nodes opsplijt in verschillende hidden layers. Wij hebben het zo ingesteld dat er 2 hidden layers zijn, elk met 64 nodes.

De Env is ofwel een OpenAi Gym ⁷ Environment, ofwel een SubprocVecEnv. Dit laatste is een wrapper rond een Gym Environment, maar staat Stable Baselines 3 ⁶ toe om ons PPO algoritme over verschillende threads uit te voeren. Hierbij worden verschillende stappen over verschillende threads uitgevoerd, en versnelt dus het trainingsproces.

2.2.2.2. OpenAI Gym Environments⁷

Een OpenAi Gym Environment is een interface die door verschillende Reinforcement Learning Libraries gebruikt wordt om de omgeving (voor ons het spel) voor te stellen waarop we willen een agent trainen.

Om aan deze interface te voldoen, moeten we 3 functies implementeren.

1. `init()`
2. `reset()`
3. `step()`

We bekijken elk van deze in detail.

2.2.2.2.1. Initialisatie

De `init` wordt door de library opgeroepen in het begin van de training. Deze heeft als doel om de environment en zijn variabelen te initialiseren, maar nog belangrijker om de action space en de observation space in te stellen.

De action space definieert de mogelijke acties die de agent kan uitvoeren. In ons geval is hier geen ambiguïteit aan: De 4 mogelijke acties van Pac-Man zijn een stap zetten in elk van de 4 windrichtingen.

De observation space definieert hoe een observation eruit ziet. Een observation is de verzameling van variabelen waarnaar de agent moet kijken om een actie aan te raden. Deze space is echter veel moeilijker optimaal te definiëren dan de action space. We hebben deze space dan ook grondig beëxperimenteerd en onderzocht.

We zijn begonnen met letterlijk alle variabelen in deze space te steken. Dit omvatte dus: elke tile van de map, de positie van Pac-Man, de positie en richting van de geesten, hoe lang een geest nog verwijderd is van de map nadat deze verslagen was, hoe lang we nog in de power state zitten, wat onze score is, wat onze combo is (zorgt voor hogere score als we verschillende geesten snel na elkaar verslaan) en hoeveel pellets er nog over zijn. Dit leek ons het totaalplaatje, waar het model dus het meeste linken kan ontdekken. Het bleek echter al snel dat dit te veel parameters waren. Door waarschijnlijk een te veel aan overbodige variabelen, trainde de agent trager dan verwacht, en presteerde hij ook slechter dan verwacht.

Als volgende stap hebben we het andere uiterste onderzocht. We hebben zo weinig mogelijk

variabelen meegegeven, enkel degene waarvan we zelf dachten dat ze belangrijk waren in het maken van een keuze. Op deze manier zou volgens ons de agent heel snel duidelijke linken kunnen leggen. Deze variabelen waren dan: de positie van Pac-Man, de positie van de geesten (dus niet meer de richting), hoe lang we nog in de power state zitten, en elke tegel van de map die geen muur is. Het meegeven van muren leek ons overbodig, aangezien binnen dezelfde training deze constant zijn en dus de keuze niet beïnvloeden. Al snel bleek dat onze agent niet alleen sneller leerde, ook het eindresultaat was beduidend beter. Het bleek dat deze variabelen dus een betere keuze waren.

Echter, na verder trainen en onderzoeken, bleek dat het weglaten van bepaalde variabelen voor problemen kon zorgen. Zo zou de agent waarschijnlijk te veel vanbuiten leren en zich niet snel aanpassen, en dus door slecht geluk toch kunnen slecht presteren. Aangezien het vanbuiten leren van een map niet de bedoeling is van een agent, en omdat we liever toch nog een iets stabielere resultaat verlangen, hebben we nog een laatste onderzoek gedaan naar deze observation space. We zijn uiteindelijk uitgekomen op een mooie middenweg tussen onze vorige observation spaces. Zo hebben we besloten de muren terug op te slaan, voor moesten we ooit onze code uitbreiden om op verschillende mappen tegelijk te trainen. Ook hebben we de richting van ghosts terug opgeslagen. Onze finale observation space bestaat nu uit: elke tile van de map, de positie van Pac-Man, de positie en richting van de geesten en hoe lang we nog in de power state zitten. Deze space produceert gemiddeld iets lagere scores dan de vorige space, maar is een stuk consistentere. Zo zijn er veel minder runs waarbij Pac-Man sterft binnen enkele stappen. Ook is er nu de mogelijkheid het onderzoek uit te breiden om te trainen op meerdere mappen.

2.2.2.2.2. Reset

Deze functie wordt door de libraries opgeroepen wanneer een run geëindigd is en een nieuwe run moet opgestart worden. Het enigste dat hier moet gebeuren is dat de environment opnieuw opgestart wordt en dat alle variabelen terug naar hun beginpositie gezet worden. Hier gebeurt niets waar er onderzoek naar nodig was.

2.2.2.2.3. Step

Deze functie is mogelijks de belangrijkste functie in een Gym environment. Hij stelt een stap in het trainingsproces voor. De library roept deze functie elke stap van zijn training op, met als argument een actie die binnen de action space ligt. De functie moet dan deze actie uitvoeren en dus de state aanpassen, en geeft dan de nieuwe observatie (gespecificeerd door de observation space) en de reward terug, en of het spel gedaan is of niet (bij ons is dit of Pac-Man verslagen is of alle pellets verzameld zijn). De reward beschrijft hoe voordelig de genomen actie was. De agent gebruikt dan deze reward om zijn neurale netwerk te updaten, aan de hand van het gekozen algoritme (bij ons PPO).

Deze reward is, net zoals de observation space, een belangrijke factor in het presteren van de agent. Het is dan ook vanzelfsprekend dat dit een kern van ons onderzoek was.

We zijn begonnen met een zeer simpele, straight forward, reward functie: het verschil tussen de nieuwe score en de vorige score. De reden dat we met een simpele functie begonnen zijn, is om te onderzoeken of de agent kan leren uit enkel de basis. Dit bleek al snel niet het geval. De agent stierf heel snel en was weinig geïnteresseerd in de pellets. Het voelde alsof de agent willekeurige keuzes maakte. We moesten dus duidelijk de reward functie verduidelijken als hulp voor de agent.

Het eerste probleem dat we wouden oplossen, was het snel sterven. De agent zag mogelijks sterven niet noodzakelijk als iets negatiefs. Om dit te voorkomen, was het genoeg om een grote penalty te geven bij het verlies van een spel. Na deze toevoeging, zagen we de agent al snel zetten nemen die hem weg van geesten brachten. Het oppakken van pellets bleef echter willekeurig aanvoelen. Als hij een rij pellets afgang, zou hij het pad pellets blijven volgen. Als er echter geen pellets in de directe omgeving waren, was de agent meer bezig met het ontwijken van de geesten dan het halen van de verdere pellets, waardoor er geen vooruitgang meer te zien was. De oplossing hiervoor is gelijkaardig aan hoe we het vorige probleem hebben opgelost: We geven een grote bonus als de agent erin slaagt om alle pellets op te pakken, en dus te winnen. Op deze manier zal de agent de pellets verzamelen even belangrijk vinden als het overleven.

De agent had hieraan genoeg om de regels van het spel te begrijpen en te volgen. Echter, er was nog geen strategisch inzicht opmerikbaar. De agent deed weinig domme dingen, maar deed ook weinig intelligente dingen. Er was dus nog een laatste aanpassing aan de reward functie nodig. We ondervonden al snel het probleem in onze reward: indien er niets gebeurde in een stap, was de reward 0, waaruit de agent niet veel info kan halen. Dit was zeker een probleem, aangezien in grotere mappen deze situatie uiterst veel gaat voorkomen. We moesten dus een bepaalde reward teruggeven in het geval er geen nieuwe pellets opgepakt werden en de agent niet gestorven is. Wat we moesten teruggeven, was echter een van de moeilijkste dilemma's van dit project. We konden namelijk ofwel een positieve reward teruggeven, of een negatieve reward. Deze twee mogelijkheden zijn dan wel tegenovergesteld, ze hebben beiden goede argumenten om gekozen te worden. We hebben dan ook beiden grondig onderzocht.

Onze eerste gedachtengang was dat, ook al was er geen strategisch denken door de agent, de agent uiteindelijk elke map kon winnen als hij gespecialiseerd raakte in het ontwijken van geesten. Uiteindelijk, na lang genoeg overleven, zal de agent wel alle pellets opgepakt hebben. Met deze argumenten in ons achterhoofd, hebben we dan de agent een positieve reward gegeven indien er niets gebeurd is (de reward voor het pakken van een pellet was natuurlijk hoger dan deze reward). Wat we verwachtten te gebeuren gebeurde echter niet. Tot onze verbazing ging de agent gewoon stilstaan in een hoek, bij elke run dezelfde. Als er een geest in de buurt kwam, zou hij even de hoek verlaten, en achteraf terug daar gaan stilstaan. Door deze manier van geesten ontwijken, bereikte hij echter nooit de andere kant van de map, en zou het spel dus nooit eindigen. Na verder onderzoek bleek dat de agent de hoek berekende die de laagste kans heeft om bezocht te worden door een geest. We konden dus tot de conclusie komen dat, statistisch gezien, bij het verlaten van die hoek het gemiddelde scoreverlies, door het stijgen van de kans op sterven door de geesten, groter was dan de scorewinst die de pellets ons brachten. Na experimenteren met hoe groot elke reward is, was er weinig verbetering te merken.

Na het vorige gefaalde experiment, was het tijd om de andere mogelijkheid te onderzoeken: een negatieve reward indien er geen verschil is in

score. Net zoals de positieve reward, heeft deze mogelijkheid overtuigende argumenten. Toen ons reward nog 0 was, zagen we weinig strategisch inzicht. Indien we echter de agent een negatieve reward geven, wordt deze geforceerd om na te denken over hoe zo snel en veilig mogelijk alle pellets te verzamelen. We verplichten de agent zeg maar om actie te ondernemen, en niet doelloos rond te dwalen. Deze strategie bleek al snel uiterst goed te werken. Bij grotere maps werd al na redelijk korte trainingen grote delen van de map opgepakt voor de agent stierf. Meer nog, niet alleen kon de agent goed pellets verzamelen, hij was tegelijk ook goed in het ontwijken van de geesten. Dit was het strategisch inzicht dat we verwachtten van de agent. Dit brengt ons dan tot onze finale reward functie:

```
prev_score <- game.score
update_game()
reward <- game.score - prev_score
if game.pellets is 0:
    reward <- grote bonus
else if game.is_over:
    reward <- grote penalty
if reward is 0:
    reward <- kleine penalty
return reward
```

2.3. Optuna⁴

Eens we PPO opgezet hadden, hadden we echter nog een probleem. De trainingen waren niet zo succesvol als we hoopten, en al snel bleek dit te komen door een slechte keuze aan hyperparameters. Na enige tijd zelf onderzoek doen naar goede hyperparameters, bleek dit veel moeilijker te zijn dan verwacht, en maakten we maar weinig vooruitgang in hoe succesvol de trainingen waren. We zijn hiervoor op zoek gegaan naar een oplossing en kwamen op het idee om hier een framework voor te gebruiken dat dit automatisch zal doen. Optuna is zo een framework.

We bekijken hoe Optuna in elkaar zit en hoe we het gebruikt hebben.

2.3.1. Achtergrond Optuna

Traditioneel werd automatische hyperparameter optimalisatie gedaan aan de hand van oftewel grid search oftewel random search algoritmes. Grid search is een brute force algoritme dat alle mogelijke combinaties in een grid afgaat en random search zal random samplen en het beste resultaat teruggeven. Deze twee methoden zijn beide heel

resource-intensive. In de laatste jaren zijn er echter verschillende soorten algoritmen ontwikkeld om dit op een betere manier te doen.

Optuna is een bibliotheek die verschillende van deze sampling algoritmen implementeert. Daarbovenop biedt optuna ook nog een extra mechanisme om oplossingsruimten vroegtijdig te stoppen aan de hand van een vooraf bepaald criterium. Dit mechanisme noemt men pruning. De combinatie van deze 2 technieken maakt het mogelijk om heel efficiënt de beste parameters te gaan vinden.

Voor het samplen gebruikt optuna standaard een Bayesian optimization algorithm, gebruik makend van een Tree-structured Parzen Estimator (TPE). Dit is ook het algoritme waar wij voor kozen. Optuna heeft verder ook nog implementaties van andere sampling strategieën zoals een *NSGAII Sampler*, *CMA-ES Sampler*, *MOTPE Sampler*,...

De TPE wordt gebruikt voor een sequentieel model-gebaseerde optimalisatie (SMBO) methode. Dit betekent dat sequentieel modellen gemaakt worden die de performantie van de hyperparameters benaderen op basis van historische metingen en vervolgens nieuwe parameters gekozen worden op basis van deze modellen.

De TPE benaderd modellen $P(x|y)$ en $P(y)$. Hierbij representeert x de hyperparameters en y de geassocieerde kwaliteits score. $P(x|y)$ wordt gemodelleerd door hyperparameters te transformeren, door de verdelingen van de vorige configuratie te vervangen door niet-parametrische dichtheid.

Meer informatie hieromtrent zullen we niet geven. Voor meer informatie kan u de references bekijken.

2.4. Gebruik van Optuna

Het gebruik van optuna bestaat uit 2 delen. Het eerste deel is een Optuna studie en het tweede deel is de object functie die we aan de study meegeven.

De studie aanmaken doen we als volgt:

```
study = optuna.create_study(
    direction = "maximize")
```

Hierna roepen we de optimize functie op van de studie. Deze vraagt de objective function.


```
study.optimize(
    objective_function,
    n_trials=TUNING_STEPS,
    gc_after_trial=True)
```

Hierbij is `objective_function` de objective functie die we meegeven, `TUNING_STEPS` het aantal keer dat we de hyperparameters updaten en tot slot `gc_after_trial=True` zet garbage collection aan zodat we niet zonder geheugen komen te zitten. Dit laatste zal het process een beetje vertragen maar vermits we anders zonder geheugen komen te zitten, hebben we geen andere keuze dan dit aan te zetten.

Nu we de optuna studie bekeken hebben, bekijken we ook de objective functie. De objective functie zal voor elke update van de hyperparameters opgeroepen worden. De objective functie bevat drie belangrijke delen. Ten eerste krijgt deze een `trial` argument mee van de optuna studie. Vervolgens moeten we de hyperparameters die optuna moet optimaliseren selecteren. Tot slot moeten we de metric die optuna moet maximaliseren bepalen en teruggeven.

Hyperparameters selecteren doen we als volgt. We kijken bijvoorbeeld naar het selecteren van de `learning rate`.

```
lr = trial.suggest_float(
    'learning_rate',
    lowerbound,
    upperbound)
```

Hier zien we dat we een float selecteren, de naam `'learning_rate'` aan de parameter geven en een lower en upper bound selecteren voor de hyperparameter.

Tot slot moeten we ook nog een metric teruggeven die we willen optimaliseren. Hiervoor werd er gekozen om de gemiddelde reward terug te geven van 10 episodes van het nieuwe policy, die getraind was voor een middelgroot aantal tijdstappen met de gekozen hyperparameters. We doen dit aan de hand van de `evaluate_policy(model, env)` functie die stable baselines⁶ ons levert.

We bekomen dan de volgende functie:

```
OBJECTIVE_PPO_PACMAN
in: trial
1. select hyperparamters
2. env ← new pacman environment
3. model ← PPO(hyperparameters...)
4. model.learn(...)
5. mean_reward ← evaluate_policy(
    model, env)
6. return mean_reward
```

2.5. Globaal Overzicht Trainings Process

Nu elk deel van onze code en ons onderzoek overlopen is, is het tijd om te bespreken hoe onze agent effectief getraind wordt.

We beginnen met het bepalen van enkele configuratie parameters. Zo beslissen we voor elke training hoeveel stappen Optuna moet uitvoeren, hoe lang er getraind moet worden per stap die Optuna uitvoert, en hoe lang onze echte training zal duren. Deze parameters bepalen de stabiliteit, de totale uitvoeringstijd en de effectiviteit van het trainen.

Daarnaast bepalen we ook de lower- en upper bounds van elke hyperparameter die we willen tunen. Aangezien vele hyperparameters slechts een korte range aan logische, behulpzame waardes hebben, kan zo Optuna sneller aan goede resultaten komen.

Hierna begint de Optuna study, die dus een vooraf gespecificeerd aantal stappen gaat uitvoeren. Bij elke stap, zal een nieuw Pac-Man environment en nieuw model gecreeerd worden, en zal via PPO op deze environment voor een vooraf gespecificeerde tijd getraind worden, met de voorgestelde hyperparameters. Na deze training worden er 10 runs uitgevoerd op het getrainde model, en wordt de gemiddelde score van deze runs berekend. Dit gemiddelde benadert dan hoe goed de door Optuna voorgestelde hyperparameters waren. Hierna begint Optuna aan de volgende stap en kiest het nieuwe hyperparameters.

Als de Optuna study al zijn stappen gedaan heeft, geeft deze ons terug welke hyperparameters tot de hoogste gemiddelde score hebben geleid. Hierna voeren we dan de effectieve training uit. We trainen met PPO de agent voor een vooraf gespecificeerde tijd (die veel langer is dan de tijd van het trainen per Optuna stap), met deze beste hyperparameters. Eens deze training beëindigd is, hebben we onze getrainde agent. Deze laten we

dan runs uitvoeren aan de hand van onze terminal output, zodat we met eigen ogen kunnen zien hoe goed deze agent presteert, en of hij bepaalde strategieën heeft ontdekt. We schrijven onder-tussen ook de score en de overlevingstijd per run uit naar een bestand, dat we dan achteraf kunnen analyseren.

3. Resultaten

Nu we de code en het trainingsproces besproken hebben, wordt het tijd om deze training effectief toe te passen op elk level en te kijken hoe de agent zich gedraagt. We analyseren per getrainde agent 1.000 runs, en bekijken hier dan verschillende statistieken op, zoals de score en de overlevingstijd.

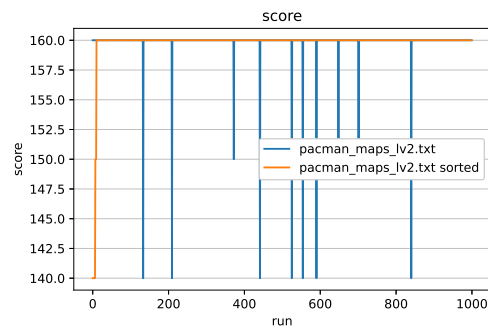
3.1. Klein Level

Het is een logische eerste stap om te beginnen bij een heel klein level. Dit level ziet er als volgt uit:

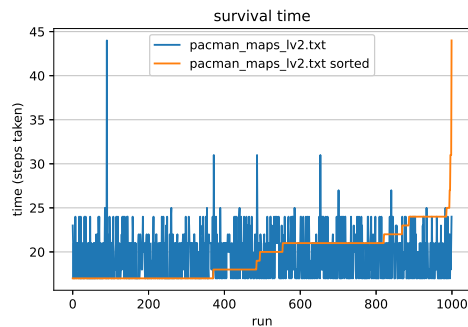
```
#####
#P.....#
#.#.#.#.#
#.....G#
#####
```

In deze map hoort de agent nog maar weinig inzicht te hebben. We verwachtten dat heel snel een perfecte score zou gehaald worden. Dit bleek dan ook te kloppen na een relatief kleine trainingstijd van slechts 1.000.000 stappen.

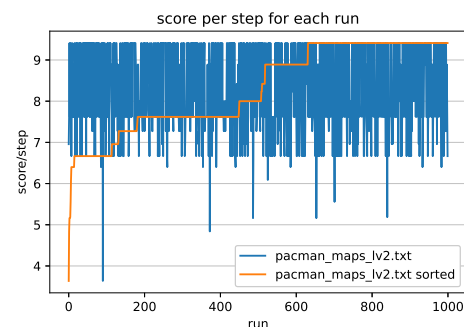
Na 1.000 runs met deze getrainde agent, had de agent een winpercentage van 99%. De maximum te behalen score op dit level is 160 (16 pellets die elk 10 punten waard zijn, waarbij onder een geest altijd een pellet staat). De gemiddelde score van deze 1.000 runs bedraagde 159.83 en het gemiddeld aantal stappen van een run is 19.574. Dit ligt zeer dicht bij het effectieve minimum in stappen om te winnen, 17. De agent presteert zo goed als perfect.



Op deze grafiek kan u zien wat de score van elke run was. Zoals u ziet heeft het merendeel van de runs een maximumscore van 160 en zijn er slechts enkele mindere runs met een score die nog steeds zo goed als perfect is.



Op deze grafiek ziet u hoeveel stappen een run heeft geduurd voor deze ofwel verloren ofwel gewonnen is. Opvallend aan deze grafiek is dat de runtijd vrij constant is en er slechts een heel klein aantal runs zijn die opmerkelijk langer duren dan de andere runs. We veronderstellen dat in deze runs de agent in een ronde aan het lopen is.



In deze grafiek kan u zien wat de gemiddelde scorewinst per stap is van een run. Hier kunt u heel duidelijk opmerken dat een deel van de runs optimaal presteert. Slechts een heel klein deel heeft hierop lage waarden, en negeert dus de pellets.

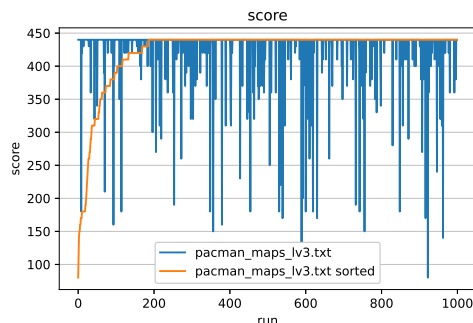
3.2. Middelgroot Level

Nadat dit level, zonder veel verbazing, een succes was, hebben we vervolgens een agent getraind op een level van middelmatige grootte. Deze ziet er als volgt uit:

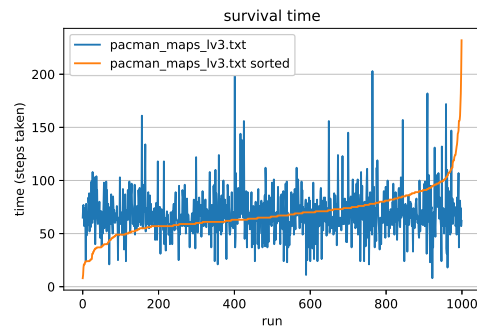
```
#####
#P.....#
#.#####.
#.#####.
#.....#
#.#####.
#.....G#
#####
```

Deze map stelt een extensie voor op de kleine map. Hier heeft de agent meer keuzes om te maken en wordt er nog beter getest of hij de pellets probeert te volgen. We verwachtten nog steeds een zeer goede prestatie, met mogelijks een winpercentage dat terug tegen de 100% grenst. Dit bleek ongeveer te kloppen na een training van 2.000.000 stappen.

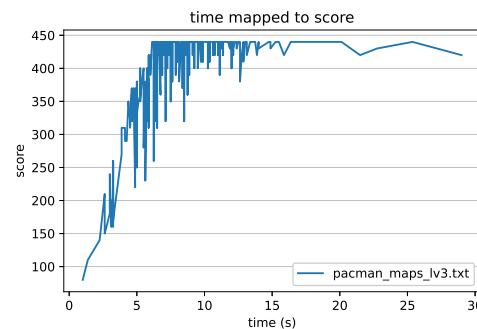
De maximum score voor deze map bedraagt 440. De agent behaalt een winpercentage van 81.5% en een gemiddelde score van 423.77, wat opnieuw heel dicht in de buurt ligt van perfect presteren.



Op deze grafiek kan u zien dat, ook al zijn niet alle runs perfect, hoe kleiner de score is, hoe kleiner de kans op deze score is.



Hier ziet u een vorm die u bij de rest van de overlevingstijd-grafieken zal herkennen. Deze vorm betekent dat de meeste runs een bepaalde begrensde tijd in beslag nemen. Er zijn slechts enkele runs die onmiddellijk sterven en dus te agressief pellets opzoeken en hierdoor de geesten niet ontwijken. Er zijn echter ook slechts enkele runs die heel lang duren, en dus niet genoeg aandacht aan de pellets besteden. Hieruit kunnen we afleiden dat onze training stabiele resultaten heeft die een goede balans zijn tussen pellets opnemen en geesten ontwijken.

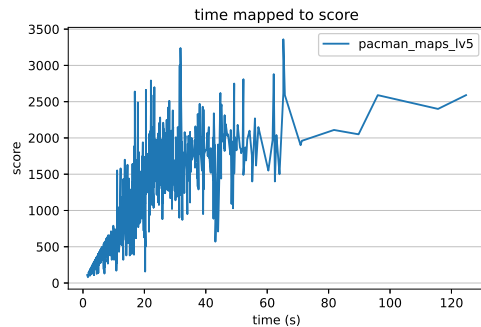


Deze laatste grafiek toont u de relatie tussen de uitvoeringstijd en de behaalde score. Uit deze grafiek kunnen we afleiden dat een perfecte score na slechts 6 a 7 seconden kan behaald worden. We zien ook dat bij de hele lange runs, deze perfecte score niet altijd gehaald wordt. We veronderstellen dat de agent hier dan vast zit in een loop en hier niet uit geraakt, ook al komt de geest op zijn pad.

3.3. De originele map

Nu we weten dat onze agent goed presteert op de kleinere mappen, vonden we het tijd om de agent te trainen op de belangrijkste map: de map van het originele spel. Deze map heeft de volgende lay-out:

heeft gevonden tussen pellets verzamelen en geesten ontwijken.



Tot slot hebben we deze grafiek, die per run de score op de runtijd afbeeldt. Hier ziet u een duidelijk stijgende progressie. Hoe langer een run dus duurt, hoe meer punten behaald worden. Dit is logisch, maar ook geruststellend, aangezien dit wijst op het feit dat lange runs gelijk zijn aan goede runs, en niet gewoon aan runs waarbij de agent vast zit in een deel van de map.

4. Conclusie

Nu al de resultaten gekend zijn, kunnen we de onderzoeksvragen beantwoorden.

Kan Pac-Man winnen? Op mappen van kleinere schaal kan Pac-Man winnen, zelfs met hoge slaagpercentages. Op de complexere en grotere mappen slaagt de agent hier jammergenoeg niet in.

Welke strategieën ontdekt Pac-Man? Op onze finale trainingen, had de agent een heel specifieke, ietwat succesvolle, strategie ontdekt. Deze is om in een hoekje te wachten, een pad te berekenen dat de hoogste score oplevert en dat het veiligst is, en te wachten tot dit pad vrij is, om vervolgens dit pad af te lopen en het process te herhalen. Een andere interessante strategie van eerder in het onderzoek, die de agent ontdekte toen onze reward nog positief was indien er geen scoreverschil is, was om te berekenen welke hoek de kleinste kans heeft om bezocht te worden, en dan in deze hoek stil te staan en eventueel kleine ontwikkelingsmanoeuvres te maken in geval van gevaar.

Welke invloed hebben het aantal spoken en de map lay-out op de vorige onderzoeksvragen? Het aantal spoken heeft een enorm effect op de prestatie van de training. Hoe meer spoken, hoe meer variantie, hoe meer de agent onverwachts zal sterven. De map-layout zelf heeft minder een effect. Wat echter wel nog een effect heeft, is de aanwezigheid van power pellets. Blijkbaar kan de

agent beter presteren en sneller een goede strategie vinden indien er power pellets aanwezig zijn in een level. Volgens ons komt dit omdat power pellets de agent een vorm van controle geven, alsook een resistentie tegen de hoge variantie van de spoken.

Al zijn er interessante dingen op te merken op de uiteindelijke map, merken we op dat de agent geen enkele keer kan winnen en de agent dus niet werkt zoals verwacht op mappen met enige ingewikkeldheid. Dit resultaat is teleurstellend, maar gelukkig kan de agent perfect zijn baan vinden in de kleinere mappen. Er kunnen verschillende redenen zijn waarom onze agent ondermaats presteert:

1. Te groot verschil tussen de trainingstijd per Optuna stap en de effectieve trainingstijd. De Optuna study optimaliseert mogelijks parameters die beter werken op korte trainingen dan op lange trainingen. Bij een te groot verschil in grootte kan dit nadelige gevolgen hebben. Dit zou opgelost kunnen worden door een langere uitvoeringstijd.
2. Te korte effectieve training. Er zijn heel veel variabelen en mogelijke observaties waardoor mogelijks het model niet genoeg tijd heeft gekregen om verbanden te vinden. Ook dit zou kunnen opgelost worden door een verlenging van de trainingstijd.
3. Te weinig geëxperimenteerd met verschillende netwerkstructuren. We hebben vrij snel in het onderzoek besloten om te werken met 2 hidden layers met elk 64 nodes. Hiernaar konden we, indien er meer tijd tot onze beschikking was, meer en uitbundiger onderzoek gedaan hebben.
4. Te weinig variantie. De kans bestaat nog steeds dat de agent de mappen te veel vanbuiten probeert te leren en daardoor snel sterft door een geest in zijn pad. Dit kon opgelost worden door de agent te trainen op willekeurige mappen, waarbij zowel Pac-Man als de geesten op willekeurige plaatsen starten. Hierdoor zou de agent mogelijks sneller linken leggen tussen deze entiteiten. Moesten we dit onderzoek opnieuw kunnen beginnen, zouden we deze eigenschappen van in het begin geïmplementeerd hebben.
5. Te weinig tussenresultaten en documentatie opgeslagen. We hebben op sommige momenten in ons onderzoek dezelfde training meerdere keren moeten uitvoeren nadat we andere verwachtingen hadden van de run. Zo

moesten we lange trainingen opnieuw uitvoeren om de grafieken te construeren die u in dit verslag terugvindt, aangezien we bij de vorige trainingen geen data van 1.000 runs hebben opgeslagen. Indien we het onderzoek opnieuw kunnen beginnen, hadden we langer stilgestaan bij welke data we nodig hebben voor het onderzoek, waardoor we meer tijd zouden hebben gehad om andere dingen te onderzoeken.

We zijn verder ook tot de conclusie gekomen dat Reinforcement Learning algoritmes nog steeds vrij onstabiel zijn. Het is bij het begin van een training altijd moeilijk in te schatten of dat de training goede resultaten zal opbrengen of dat deze resultaten zal opbrengen die even goed zijn als die van een veel kortere training. Dit komt deels door de vele interagerende parameters. Er zijn heel veel parameters en keuzes met veel nuance die de effectiviteit bepalen van een training, wat het heel moeilijk maakt deze allemaal afzonderlijk te onderzoeken en te tunen.

We hebben beiden het gevoel dat we dankzij dit project heel veel hebben bijgeleerd over AI en onderzoek ernaar doen. We zijn tot nuttige reflecties gekomen die we bij volgende onderzoeken zeker zullen toepassen. We hopen dat er nog veel onderzoek gebeurt aan de hand van Reinforcement Learning algoritmes, aangezien we beiden deze algoritmes enorm interessant vinden. We kijken uit naar de toekomst van AI!

References

1. Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Graves, Alex, Antonoglou, Ioannis, Wierstra, Daan, and Riedmiller, Martin, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602* (2013).
2. Schulman, John, Levine, Sergey, Abbeel, Pieter, Jordan, Michael, and Moritz, Philipp, "Trust region policy optimization" in *International conference on machine learning*, pp. 1889-1897, PMLR (2015).
3. Schulman, John, Wolski, Filip, Dhariwal, Prafulla, Radford, Alec, and Klimov, Oleg, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347* (2017).
4. Akiba, Takuya, Sano, Shotaro, Yanase, Toshihiko, Ohta, Takeru, Koyama, and Masanori, "Optuna: A Next-generation Hyperparameter Optimization Framework" in *Proceedings of the 25th {ACM} {SIGKDD} International Conference on Knowledge Discovery and Data Mining* (2019).
5. Shlens, Jonathon, "Notes on kullback-leibler divergence and likelihood," *arXiv preprint arXiv:1404.2000* (2014).
6. Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann, "Stable-Baselines3: Reliable Reinforcement Learning Implementations," *Journal of Machine Learning Research* **22**, pp. 1-8 (2021). <http://jmlr.org/papers/v22/20-1364.html>.
7. Brockman, Greg, Cheung, Vicki, Pettersson, Ludwig, Schneider, Jonas, Schulman, John, Tang, Jie, and Zaremba, Wojciech, "Openai gym," *arXiv preprint arXiv:1606.01540* (2016).