Využití metod umělé inteligence pro řešení populárních mobilních her

Lukáš Záruba

Bakalářská práce
2024

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2023/2024

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Lukáš Záruba**
Osobní číslo:	**A20647**
Studijní program:	**B0613A140020 Softwarové inženýrství**
Forma studia:	**Prezenční**
Téma práce:	**Využití metod umělé inteligence pro řešení populárních mobilních her**
Téma práce anglicky:	**Artificial Intelligence Methods for Solving Popular Mobile Games**

Zásady pro vypracování

1. Zpracujte literární rešerši na dané téma.
2. Analyzujte existující přístupy pro automatizované hraní mobilních her.
3. Vybrané metody implementujte a ozkoušejte na populárních hrách.
4. Vyhodnoťte výsledky jednotlivých přístupů z hlediska úspěšnosti řešení.
5. Vytvořte propagační přehled využívaných technik na populárních hrách.

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. STONE, James V. *Artificial intelligence engines: a tutorial introduction to the mathematics of deep learning*. USA: Sebtel Press, 2019. ISBN 978-0-9563728-1-9.
2. ROTHMAN, Denis; LAMON, Matthew; KUMAR, Rahul; NAGARAJA, Abishek; ZIAI, Amir et al. *Python: Beginner's Guide to Artificial Intelligence: Build Applications to Intelligently Interact with the World around You Using Python*. Packt Publishing, 2018. ISBN 978-178995732X.
3. GOODFELLOW, Ian; BENGIO, Yoshua a COURVILLE, Aaron. *Deep learning. Adaptive computation and machine learning series*. Cambridge, MA: MIT press, [2016]. ISBN 978-0262035613.
4. AGGARWAL, Charu C. *Neural networks and deep learning: a textbook*. 1. Cham: Springer, [2018]. ISBN 978-3-030--06856-1.

Vedoucí bakalářské práce: **Ing. Adam Viktorin, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání bakalářské práce: **5. listopadu 2023**
Termín odevzdání bakalářské práce: **13. května 2024**

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
DĚKAN
L.S.

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 5. ledna 2024

Prohlašuji, že

- beru na vědomí, že odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – bakalářskou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne

Lukáš Záruba, v.r.
podpis studenta

ABSTRAKT

Bakalářská práce zkoumá využití metod umělé inteligence (AI) pro řešení populárních mobilních her. Cílem práce je analyzovat a implementovat vybrané metody AI na vybrané mobilní hry.

Teoretická část práce se zabývá stručným popisem historie AI a rozsáhlejším popisem metod AI pro řešení mobilních her, jako jsou neuronové sítě, posilující učení a genetické algoritmy. Následuje popis možností a limitů AI v mobilních hrách a etické aspekty.

Praktická část stručně popisuje hry Flappy Bird a Snake a jejich herní principy. Implementaci NEAT algoritmu a metody Q-learningu včetně popisu herního prostředí a parametrů učení hry Flappy Bird a Snake. Testování implementace, kde je popis, jak metody fungují v jednotlivých hrách, a nakonec analýza výsledků ve formě grafů a statistik v terminálu.

Klíčová slova: Umělá inteligence, mobilní hry, NEAT algoritmus, Q-learning, Flappy Bird, Snake

ABSTRACT

The bachelor thesis explores the application of artificial intelligence methods for solving popular mobile games. The aim of the thesis is to analyze and implement selected AI methods on specific mobile games.

The theoretical part of the thesis provides a brief overview of the history of AI, followed by a more detailed description of AI methods for solving mobile games, such as neural networks, reinforcement learning, and genetic algorithms. The possibilites and limitations of AI in mobile games and the ethical aspects are also discussed.

The practical part briefly describes the games Flappy Bird and Snake, along with their gameplay principles. It then delves into the implementation of the NEAT algorithm and the Q-learning method, including a description of the game environment and learning parameters for both Flappy Bird and Snake. The implementation is tested, and the methods functionality in each game is explained. Finally, the results are analyzed and presented in the form of graphs and statistics in the terminal.

Keywords: Artificial intelligence, mobile games, NEAT algorithm, Q-learning, Flappy Bird, Snake

PODĚKOVÁNÍ

Vyjadřuji hlubokou vděčnost panu Ing. Adamu Viktorinovi, Ph.D, mému vedoucímu bakalářské práce, za jeho vedení, vstřícnost a cenné rady, které vedly k úspěšnému dokončení této práce.

V neposlední řadě bych rád poděkoval mé rodině a přátelům za jejich podporu a povzbuzování během celého studia.

OBSAH

ÚVOD

V současném digitálním věku se mobilní hry staly nepostradatelnou součástí každodenního života milionů uživatelů po celém světě. S rostoucím zájmem o herní zábavu se však zvyšují i nároky na kvalitu a interaktivitu her, což klade velký tlak na vývojáře, aby vytvářeli stále sofistikovanější a zábavnější herní zážitky. Umělá inteligence se stává nedílnou součástí herního průmyslu, poskytující klíčové nástroje pro dosažení vysoké úrovně realismu, dynamiky a interaktivity.

Cílem této bakalářské práce je prozkoumat a analyzovat různé metody umělé inteligence v kontextu populárních mobilních her. Důraz bude kladen na teoretické základy, jako je historie umělé inteligence a metody, které jsou často využívány v herním průmyslu. Tato práce se také zaměří na praktickou aplikaci těchto metod prostřednictvím implementace algoritmů NEAT a Q-learningu ve dvou ikonických hrách – Flappy Bird a Snake.

První část se věnuje rozboru teoretických základů umělé inteligence. Historie umělé inteligence je sledována od svých počátků až po současnost. Dále budou probírány různé metody umělé inteligence, které nacházejí uplatnění v herním prostředí. Diskutovány budou také možnosti a limity umělé inteligence v mobilních hrách. Po této diskusi budou rozebrány etické aspekty využívání umělé inteligence v herním prostředí..

Druhá část práce se soustředí na konkrétní aplikaci získaných poznatků a metod v praxi. Prostřednictvím implementace NEAT algoritmu ve hře Flappy Bird a využitím Q-learningu pro optimalizaci hraní hry Snake budou demonstrovány možnosti a úskalí použití umělé inteligence v reálných herních scénářích.

I. TEORETICKÁ ČÁST

1 UMĚLÁ INTELIGENCE

Tato kapitola se zabývá přehledem vývoje umělé inteligence (AI) od počátku do současnosti.

1.1 Historie umělé inteligence

Padesátá léta 20. století jsou považována za zrození oboru AI. V tomto desetiletí se odehrály klíčové události, které položily základy pro další vývoj AI. V roce 1950 Alan Turing, britský matematik a informatik, navrhl Turingův test jako způsob, jak posoudit inteligenci stroje. Test je založen na myšlence, že pokud stroj dokáže vést konverzaci s člověkem tak, že ho nerozezná od jiného člověka, pak lze stroj považovat za inteligentní. [1][2] Dartmouthská konference, která se konala na Dartmouth College v USA v roce 1956, je považována za oficiální zrod oboru AI. Shromáždila klíčové vědce a filozofy, kteří diskutovali o možnostech strojového myšlení a učení. V roce 1958 John McCarthy, americký informatik, vyvinul jazyk Lisp, který je dnes druhým nejstarším programovacím jazykem na vysoké úrovni a který je dodnes používán v oblasti AI. Lisp je speciálně navržen pro manipulaci se symboly a logickými strukturami, což je klíčové pro symbolické uvažování, které dominovalo v rané fázi AI. [1]

Šedesátá léta byla pro výzkum AI obdobím velkého optimismu a rychlého pokroku. Toto období, často označované jako „zlatý věk AI“, bylo svědkem zavedení mnoha klíčových konceptů a technik které tvoří základ moderní AI. Symbolické uvažování, které se zaměřuje na manipulaci se symboly a pravidly pro reprezentaci a řešení problémů, bylo dominantním přístupem v 60. letech. V tomto období vznikly programy jako General Problem Solver (GPS) a ELIZA, které demonstrovaly schopnost strojů řešit logické úkoly a simulovat konverzaci. ELIZA vytvořená v roce 1966 Josephem Weizenbaumem z MIT, je považována za jednoho z prvních chatbotů a průkopníka v oblasti konverzační umělé inteligence. [1][3] Mezi další oblasti výzkumu v 60. letech patřily neuronové sítě, které byly inspirovány strukturou a funkcí lidského mozku. Vědci jako Frank Rosenblatt a Bernard Widrow vyvinuli základní modely neuronových sítí a prokázali jejich potenciál pro učení a rozpoznávání vzorů.

Sedmdesátá a osmdesátá léta pro oblast AI představovala období plné kontrastů. Na jedné straně se jednalo o období pozoruhodného pokroku, kdy se technologie AI staly výkonnějšími a sofistikovanějšími. V roce 1972 byl na Stanfordské univerzitě v Kalifornii vyvinut průkopnický expertní systém a program umělé inteligence MYCIN. Zaměřoval se na

diagnostiku a léčbu závažných infekcí způsobených bakteriemi, jako je bakteriémie a meningitida.[4] Na druhou stranu toto období poznamenala i vlna skepticismu a zklamání. Nedostatek výpočetního výkonu a dat dále bránil dosažení plného potenciálu AI. Tyto faktory, spolu s neschopností vyřešit některé ze základních problémů AI, vedly k období stagnace a poklesu zájmu o tuto oblast, které je známo jako „zima AI“. [1][2]

V devadesátých letech došlo k dramatickému zvýšení výpočetního výkonu a dostupnosti dat, což vedlo k významným průlomům v oblastech jako je strojové učení, počítačové vidění a robotika. Roku 1997 počítač Deep Blue od IBM porazil úřadujícího mistra světa v šachu Garyho Kasparova. To byl významný milník, který ukázal potenciál AI soupeřit s lidmi v komplexních strategických hrách. [1][5] V roce 1998 Google založil PageRank, algoritmus pro hodnocení webových stránek, který je založen na principech strojového učení. PageRank se stal základem vyhledávače Google a měl hluboký dopad na způsob, jakým lidé přistupují k informacím na internetu. [6]

Desetiletí 2000-2010 bylo pro oblast AI obdobím transformace a růstu. Algoritmy strojového učení, jako jsou neuronové sítě, podpůrné vektory a naivní Bayesovy klasifikátory, se staly nezbytnými nástroji pro řešení široké škály úkolů, od rozpoznávání obrazu a zpracování přirozeného jazyka až po prediktivní modelování a robotiku. [7][8] Mooreův zákon vedl k neustálému zdvojnásobení dostupného výpočetního výkonu přibližně každých 18 měsíců. To umožnilo trénovat složitější modely strojového učení a zpracovávat větší objemy dat. [9] V tomto desetiletí zažíváme explozi průlomových událostí v oblasti AI, které mění způsob, jakým žijeme a pracujeme. Jedním z pozoruhodných příkladů je Google Voice Search, systém rozpoznávání hlasu představený v roce 2010. Díky své vysoké přesnosti a snadnému použití se Google Voice Search stal populární volbou pro interakci s chytrými telefony, reproduktory a dalšími zařízeními, čímž se stal symbolem pokroku v oblasti rozhraní člověk-počítač a zpřístupnil AI širšímu publiku. [10]

Rok 2010 znamenal pro AI bod zlomu. Algoritmy hlubokého učení, inspirované strukturou lidského mozku, dokázaly dosáhnout průlomových výsledků v oblastech jako je rozpoznání obrazu, zpracování přirozeného jazyka a strojový překlad. Tyto algoritmy pohánějí mnoho dnešních nejpopulárnějších aplikací AI, od virtuálních asistentů po autonomní auta. [2] Mezi významné události v tomto období patří založení neziskové organizace pro výzkum AI OpenAI v roce 2015, která o sedm let později spustila populárního chatbota současné doby ChatGPT. [11]

2 METODY UMĚLÉ INTELIGENCE V HERNÍM PRŮMYSLU

V herním prostředí umožňuje implementace AI NPC (nehráčské postavy) autonomní rozhodování, navigaci v herním světě a interakci s okolím, ať už se jedná o hráče, nebo jiné NPC. Za účelem dosažení co nejrealističtějšího chování NPC se v herní AI uplatňují i techniky strojového učení, které vedou k sofistikovanějším rozhodovacím procesům a přirozenějším interakcím NPC.

Mezi nejčastější metody vytváření AI patří:

1. **Finite State Machines (FSM):** Jde o definování stavů, které může AI zastávat, jako například „nečinnost“, „útok“ nebo „útěk“. Dále se definují podmínky, za kterých dochází k přechodu mezi těmito stavy.
2. **Behavior Trees (BT):** Jedná se o typ hierarchické definice stavů, která umožňuje NPC složitější chování. Princip fungování spočívá v rozdělení akcí do uzlů stromu (BT). NPC se pak rozhoduje pro provedení určité akce na základě podmínek splněných v dětských a rodičovských uzlech stromu.
3. **Decision Trees (DT):** Jedná se o nástroj pro modelování rozhodovacího procesu NPC. Systém předkládá NPC různé možnosti a DT na základě definovaných podmínek a atributů vybere tu nejvhodnější.
4. **Pathfinding:** Jedná se o algoritmus, který naplánuje pro NPC optimální cestu k dosažení cíle.
5. **Machine Learning (ML):** Metody ML, a to jak podpůrné učení, tak neuronové sítě, se používají k vytvoření co nejrealističtějšího a nejpřirozenějšího chování herní AI.
6. **Rule-based systems:** Jedná se o sadu striktně definovaných pravidel, která NPC striktně dodržuje. Tato pravidla slouží k zajištění rychlého a efektivního rozhodování NPC v herním prostředí.

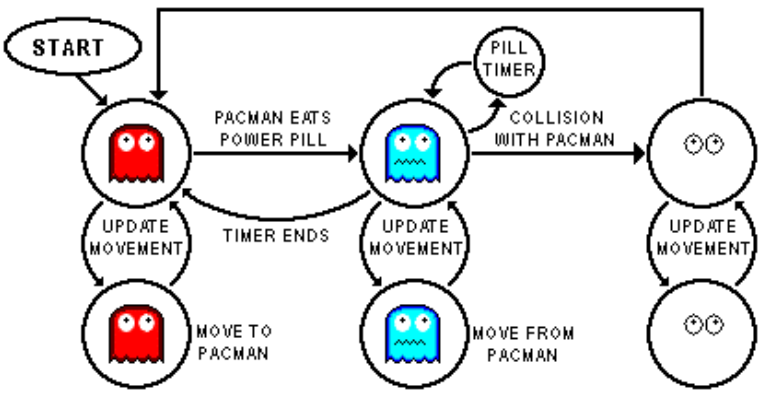
2.1 Finite state machines

Konečné automaty (FSM) představují výpočetní model, který definuje sadu stavů, jejich vzájemné přechody a akce, jež jsou nutné pro vstup do těchto stavů a výstup z nich. FSM nachází uplatnění jak v oblasti hardwaru, tak softwaru. V herním průmyslu se FSM běžně používají při tvorbě herní AI, čímž se otevírá prostor pro tvorbu rozmanitých chování a vlastností, jež AI může projevovat. [12]

Formálně jsou konečné automaty definované jako uspořádaná pětice:

- **S** je konečná neprázdná množina stavů
- **Σ** je konečná neprázdná množina vstupních symbolů
- **σ** je přechodová funkce (též přechodová tabulka), popisující pravidla přechodů mezi stavy
- **s** je počáteční stav
- **F** je množina finálních stavů

Konečné automaty se dělí na dva základní typy: deterministické a nedeterministické. Zatímco deterministický automat pro daný vstupní symbol a aktuální stav vždy striktně určí jeden následný stav, nedeterministický automat může v dané situaci přecházet do více stavů najednou. Důvodem je, že nedeterministický automat v tabulce přechodů pro daný vstupní symbol a aktuální stav nemusí mít definován pouze jeden cílový stav, ale může jich mít hned několik. Navíc je u nedeterministických automatů možné, že vstupní symbol neexistuje neboli je prázdný. V takovém případě automat automaticky přechází do všech dostupných stavů najednou. Platný vstup je pak ten, který vede alespoň do jednoho z konečných stavů. [12]



Obr. 1 Vizualizace Finite State Machines, Zdroj: https://oddwiring.com/archive/websites/mndev/MSB/GD100/fsm.htm

2.2 Behavior trees

Behavior tree, neboli česky strom chování, je specifický typ hierarchického konečného automatu. Můžeme si ho představit jako stromovou strukturu, kde jednotlivé uzly (větve) reprezentují různé akce. Pohyb v tomto stromě je řízen hierarchií a definovanými podmínkami. Začátek stromu, tzv. kořen (root), symbolizuje akci nebo cíl, kterého se snažíme dosáhnout. Podřízené uzly pak představují různé cesty, metody či podmínky, jak daného cíle dosáhnout. [13]

Při budování stromu chování se setkáváme s různými typy uzlů:

2.2.1 Uzel řízení toku

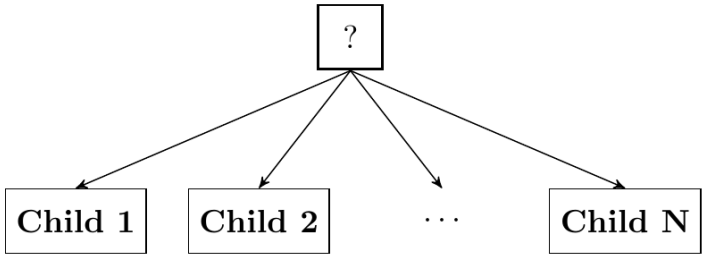
Uzel řízení toku slouží k organizaci a koordinaci podřízených uzlů v rámci stromu chování. Tyto podřízené uzly se spouštějí sekvenčně a vracejí stav "úspěch" nebo "neúspěch". Na základě těchto stavů uzel řízení toku rozhoduje o tom, zda má spustit další podřízený uzel. Mezi běžné typy uzlů řízení toku patří Selektor (Fallback), Sequence a Parallel uzel. [13][14]

Selector (Fallback) uzel

Fallback uzel, známý také jako Selector, slouží k nalezení a spuštění prvního podřízeného uzlu v seznamu, který neselže. V závislosti na výsledcích podřízených uzlů vrací Fallback uzel různé stavy: [13][14]

- **Úspěch:** Pokud alespoň jeden z podřízených uzlů vrátí stav „úspěch“, Fallback uzel vrátí stav „úspěch“.
- **Spuštěno:** Pokud všechny podřízené uzly vrátí stav „spuštěno“ (ale žádný z nich „úspěch“), Fallback uzel vrátí stav „spuštěno“.
- **Neúspěch:** Pokud všechny podřízené uzly vrátí stav „neúspěch“, Fallback uzel vrátí stav „neúspěch“.

Podřízené uzly v seznamu Fallback uzlu jsou seřazeny zleva doprava podle jejich priority.



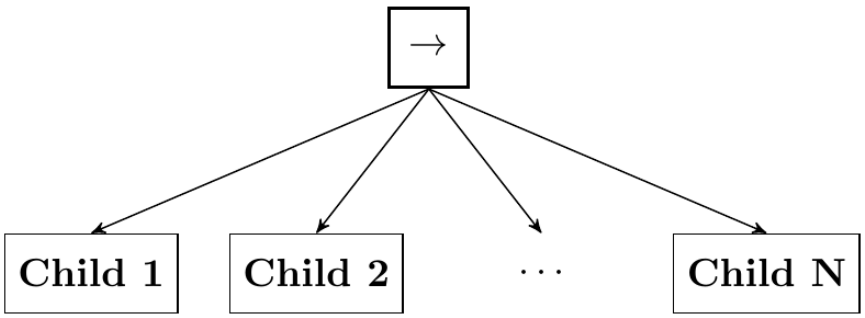
Obr. 2 Fallback uzel, Zdroj: https://engineering.nordeus.com/learning-of-behavior-trees-for-autonomous-agents/

Sequence uzel

Sequence uzel, známý také jako Sekvence, slouží k sekvenčnímu spouštění podřízených uzlů v daném pořadí. Vrací různé stavy v závislosti na výsledcích podřízených uzlů: [13][14]

- **Neúspěch:** Pokud se některý z podřízených uzlů vrátí s "neúspěchem", Sequence uzel vrátí "neúspěch".
- **Spuštěno:** Pokud všechny podřízené uzly vrátí "spuštěno" (ale žádný z nich "úspěch"), Sequence uzel vrátí "spuštěno".
- **Úspěch:** Až teprve když všechny podřízené uzly vrátí "úspěch", Sequence uzel vrátí "úspěch".

Podřízené uzly v seznamu Sequence uzlu jsou seřazeny zleva doprava podle priority. To znamená, že uzel nejprve spustí ten podřízený uzel, který je umístěn nalevo, a pokračuje v sekvenci jen v případě, že tento uzel vrátí "úspěch". Pokud se některý z podřízených uzlů vrátí s "neúspěchem", Sequence uzel se ihned zastaví a vrátí "neúspěch". [13][14]



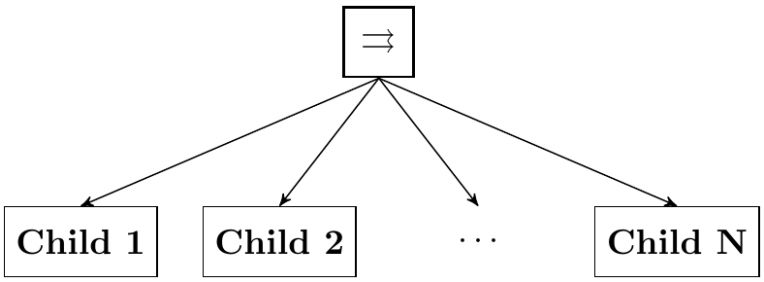
Obr. 3 Sequence uzel, Zdroj: https://engineering.nordeus.com/learning-of-behavior-trees-for-autonomous-agents/

Parallel uzel

Parallel uzel, známý také jako Paralelní uzel, spouští všechny podřízené uzly souběžně. Jeho výsledný stav závisí na dosažení určitého počtu "úspěšných" podřízených uzlů, definovaného parametrem M: [13][14]

- **Úspěch:** Pokud dosáhne M nebo více podřízených uzlů stavu „úspěch“, Parallel uzel vrátí „úspěch“.
- **Neúspěch:** Pokud více než N – M (kde N je celkový počet podřízených uzlů) vrátí stav „neúspěch“, Parallel uzel vrátí „neúspěch“.
- **Spuštěno:** V ostatních případech, kdy je dosaženo stavu „úspěch“ u méně než M podřízených uzlů, ale zároveň ne více než N – M podřízených uzlů vrátí „neúspěch“, Parallel uzel vrátí stav „spuštěno“.

Je důležité poznamenat, že pořadí, ve kterém se podřízené uzly spouštějí a vracejí své stavy, nemá vliv na výsledný stav Parallel uzlu. Dosažení požadovaného počtu "úspěšných" uzlů v libovolném pořadí splní podmínku pro návrat stavu "úspěch". Podřízené uzly v seznamu Parallel uzlu jsou seřazeny zleva doprava podle priority. To znamená, že uzel bude nejprve zkoušet spustit ten podřízený uzel, který je umístěn nalevo, a teprve poté, co ten skončí, spustí další podřízený uzel v seznamu a tak dále. [13][14]



Obr. 4 Parallel uzel, Zdroj: https://engineering.nordeus.com/learning-of-behavior-trees-for-autonomous-agents/

2.2.2 Decorator

Decorator je typ uzlu ve stromu chování, který slouží k úpravě nebo opakování výstupu podřízeného uzlu. Podřízený uzel může být pouze jeden a je povinný. Dekorativní uzel se používá k rozšíření možností a flexibility umělé inteligence. Existuje mnoho typů dekorativních uzlů, z nichž každý se zaměřuje na specifický účel. [13][14]

Mezi typy dekorátoru patří:

- **Inverter** slouží k inverzi stavu, který vrátí jeho podřízený uzel. Jinými slovy, pokud podřízený uzel vrátí "úspěch", Inverter vrátí "neúspěch", a naopak. Tento uzel se nejčastěji používá v podmínkových sekvencích, kde umožňuje snadno negaci logických podmínek. [13][14]
- **Succeeder** vždy vrací stav "úspěch" bez ohledu na stav, který vrátí jeho podřízený uzel. Jinými slovy, i když podřízený uzel selže a vrátí "neúspěch", Succeeder bude vždy indikovat, že vše proběhlo v pořádku. Tento uzel se používá v situacích, kdy očekáváme, že se u podřízeného uzlu může vyskytnout chyba, ale chceme, aby proces nadále běžel bez přerušení. [13][14]
- **Repeater** slouží k opakovanému spouštění svého podřízeného uzlu, dokud tento uzel nevrátí jakýkoliv stav. Opakování se provádí, dokud není dosaženo definovaného limitu opakování, pokud je nastaven. Pokud limit opakování není definován, Repeater bude podřízený uzel spouštět neomezeně, dokud nevrátí stav. [13][14]
- **Repeat Until Fail** je varianta Repeater uzlu s inverzní logikou. Na rozdíl od klasického Repeateru, který opakuje podřízený uzel, dokud nevrátí jakýkoliv stav, Repeat Until Fail opakuje podřízený uzel, dokud nevrátí stav "neúspěch"**.** V tomto okamžiku Repeat Until Fail vrátí stav "úspěch" svému nadřazenému uzlu. [13][14]

2.2.3 Uzel akce

Uzel akce je základní typ uzlu v hierarchii stromu chování. Nachází se na nejnižší úrovni a představuje list stromu. Uzel akce se používá výhradně jako podřízený uzel a nemůže mít žádné vlastní podřízené uzly. Ztělesňuje specifickou akci v hierarchii a může nabývat stavů "úspěch", "neúspěch" a "spuštěno". [13][14]

2.2.4 Uzel podmínky

Uzel podmínky je další základní typ uzlu v hierarchii stromu chování. Stejně jako uzel akce se nachází na nejnižší úrovni a představuje list stromu. Uzel podmínky se používá výhradně jako podřízený uzel a nemůže mít žádné vlastní podřízené uzly. Ztělesňuje logickou podmínku, která se týká určitého stavu, akce nebo vstupu. Na rozdíl od uzlu akce, uzel podmínky nabývá pouze stavů "úspěch" a "neúspěch". [13][14]

2.3 Decision trees

Rozhodovací strom (DT) je model, který popisuje proces rozhodování v rámci AI nebo algoritmus používaný v strojovém učení. Můžeme si ho představit jako jednoduchý diagram s rozvětvenými cestami, který umožňuje AI dělat rychlá a nenáročná rozhodnutí. [15]

Jednotlivé části rozhodovacího stromu se rozdělují na:

- **Kořen** představuje počáteční uzel, ze kterého se proces rozhodování a provádění akcí odvíjí.
- **Větev** představuje jednu ucelenou cestu v rámci diagramu
- **Rozdělení** představuje jednoduchou podmínku, která má převážně pravdivou a nepravdivou část.
- **Uzel rozhodnutí** představuje klíčový prvek, který řídí tok chování a rozhodování. Odkazuje na listy nebo další uzly rozhodnutí.
- **List** představuje konečný prvek, který obsahuje odpověď, akci nebo hodnotu, která reprezentuje finální výstup dané větve.

Implementace DT v strojovém učení umožňuje průběžný růst modelu, AI si postupně buduje rozsáhlou mapu rozhodování pro daný úkol. Díky jednoduchosti struktury DT je snadné model rozšiřovat a přizpůsobovat novým datům a situacím. [15]

Pro kontrolu kvality a přesnosti DT se používají metody jako Entropy a Giniho index. Tyto metody počítají míru nečistoty v uzlech a ve výsledcích stromu, čímž indikují jeho náhodnost a efektivitu. V případě, že je DT příliš rozrostlý nebo nepřesný, existují techniky pro jeho optimalizaci. Mezi nejběžnější patří: [15]

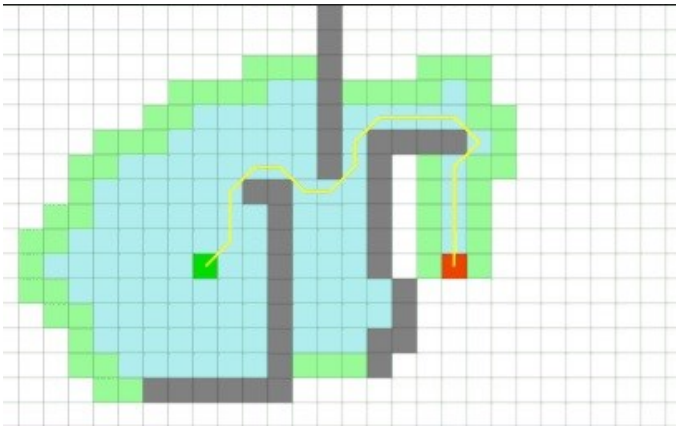
- **Pruning:** Odstranění nepotřebných větví a uzlů z DT, čímž se zjednoduší jeho struktura a zlepší se jeho výkon.
- **Boosting:** Kombinace více DT do jednoho modelu, čímž se dosáhne vyšší přesnosti a robustnosti klasifikace.

2.4 Pathfinding

Pathfinding, neboli česky hledání cesty, je klíčový proces v herní umělé inteligenci (AI), který umožňuje NPC (nehráčské postavy) procházet herní světem a dosahovat cílů. Jedná se o nalezení nejkratší **a** nejefektivnější trasy z aktuální pozice do cílového bodu. Hledání cesty je základní vlastností herní AI, která umožňuje implementaci komplexnějších herních

mechanismů a chování NPC. Díky ní můžou NPC reagovat na dynamické prostředí, vyhýbat se překážkám a efektivně plnit úkoly. Proces hledání nejkratší cesty se opírá o mnoho informací, z nichž nejdůležitější je typ algoritmu: [16]

- **Neinformované algoritmy:** Tyto algoritmy nemají přístup k úplné mapě ani k informaci o cíli. Mají k dispozici pouze lokální informace o svém okolí (např. sousední buňky) a na základě nich se rozhodují, kam se pohnout dále. [16]
- **Informované algoritmy:** Tyto algoritmy pracují s úplnou mapou herního světa a s informací o cíli. Díky tomu můžou plánovat optimální trasu s ohledem na všechny překážky a terénní vlastnosti. [16]



Obr. 5 Vizualizace Pathfinding, Zdroj: https://www.codingame.com/learn/pathfinding

2.5 Machine Learning

Strojové učení (ML) v kontextu počítačových her představuje metodu učení AI na základě informací a dat. Jedná se o rozsáhlý obor s širokou škálou aplikací v herním průmyslu, jehož cílem je zdokonalovat herní AI v různých aspektech. Hlavní oblasti využití ML v počítačových hrách: [17]

- **Trénování inteligentnějších nepřátel:** ML se používá k výcviku AI nepřátel, aby byli chytřejší, adaptabilnější a obtížněji porazitelní. Díky ML se můžou nepřátelé učit z herních situací, předvídat tahy hráče a používat různé herní strategie.
- **Vytváření realistického chování hráčů:** ML umožňuje implementovat do herní AI modely chování, které napodobují reálné hráče. To zahrnuje realistické pohyby, rozhodování, reakce na herní události a interakce s prostředím.
- **Optimalizace herního designu:** ML se dá využít k analýze herních dat a k optimalizaci herního designu. Díky ML můžou vývojáři her lépe porozumět chování hráčů,

identifikovat problematické aspekty hry a navrhovat herní mechaniky, které jsou zábavnější a poutavější.

Existuje mnoho cest vedoucích k požadovaným výsledkům, avšak mezi nejdůležitější a nejefektivnější patří tyto hlavní přístupy:

- **Učení s učitelem (Supervised learning):** Je metoda ML, která se opírá o značené datové sady. Tyto sady obsahují příklady dat (vstupy) a příslušné požadované výstupy. Algoritmus ML se na těchto datech učí, jak správně mapovat vstupy na výstupy.
- **Učení bez učitele (Unsupervised learning):** Je metoda ML, která se odlišuje od Učení s učitelem v tom, že nevyužívá značené datové sady. Místo toho se opírá o neoznačené datové sady, které obsahují pouze vstupní data bez požadovaných výstupů. Algoritmus ML se v tomto případě snaží samostatně najít strukturu a vzory v datech.
- **Zpětnovazebné učení (Reinforcement learning):** Odlišuje se od Učení s učitelem a bez učitele v tom, že se učí na základě interakce s prostředím. Algoritmus strojového učení v tomto případě neobdrží předem připravená data, ale samostatně zkoumá prostředí a učí se na základě odměn a trestů.
- **Neuronové sítě (Neural networks):** Oblast ML, která se snaží napodobit strukturu a fungování lidského mozku. Skládají se z vrstev propojených uzlů (neuronů), které zpracovávají a předávají informace. Každý uzel má váhy a aktivační funkci, které ovlivňují jeho výstup.

2.5.1 Učení s učitelem

Učení s učitelem je metoda ML, která se v herním průmyslu používá k vývoji inteligentních herních systémů. Funguje na principu trénování algoritmů na značených datových sadách. Tyto datasety obsahují vstupní hodnoty a správné požadované výstupy. Algoritmus se na nich učí, jak mapovat vstupy na výstupy s maximální přesností. [18]

Klíčové principy:

- **Trénovací data:** Klíčem k úspěchu je kvalitní a dostatečně rozsáhlý dataset s relevantními daty.
- **Ztrátová funkce:** Algoritmus se snaží minimalizovat tzv. ztrátovou funkci, která měří nesoulad mezi jeho předpovědí a požadovaným výsledkem.

- **Optimalizace:** Algoritmus se průběžně optimalizuje a zlepšuje své výsledky na základě zpětné vazby z dat.

Příklady využití v herním průmyslu:

- **Tvorba herních agentů:** Algoritmy se učí samostatně plnit úkoly v herním prostředí, jako je vyhledávání cesty, rozhodování nebo napodobování chování hráče.
- **Generování animací:** Algoritmy se učí vytvářet realistické animace postav a objektů na základě zadaných parametrů.
- **Herní analytika:** Algoritmy se používají k analýze herních dat, jako je chování hráčů, herní statistiky a trendy.
- **Anti-cheat systémy:** Algoritmy se učí rozpoznávat podvodné chování hráčů a pomáhají tak udržet fair play v online hrách.

Učení s učitelem se dále rozděluje na:

- **Klasifikační algoritmy** patří mezi základní nástroje ML, které se v herním průmyslu používají k rozpoznávání a kategorizaci dat. Jejich cílem je zařadit vstupní data do předem definovaných kategorií s co největší přesností.
- **Regresní algoritmy** jsou další důležitou oblastí ML, která se v herním průmyslu používá k analýze dat a predikci trendů. Jejich cílem je odhalit vztahy mezi závislými a nezávislými proměnnými a na základě těchto vztahů vytvářet modely pro predikci budoucích hodnot.

2.5.1.1 Naive Bayes

Naivní Bayes je klasifikační algoritmus založený na Bayesově teorii, která popisuje pravděpodobnost události na základě znalosti jiných událostí. V herním průmyslu se používá k rozpoznávání a kategorizaci dat. [18]

Princip fungování:

- **Bayesova věta**: Algoritmus používá Bayesovu větu k výpočtu pravděpodobnosti, že daný objekt patří do určitě kategorie, na základě jeho vlastností.
- **Nezávislost proměnných**: Algoritmus předpokládá, že vlastnosti objektu jsou navzájem nezávislé. To zjednodušuje výpočet, ale může vést k nepřesnostem.
- **Výpočetní rychlost**: Naivní Bayes je velmi rychlý v porovnání s jinými klasifikačními algoritmy.

2.5.1.2 Support vector machines

Support Vector Machines (SVM) je univerzální algoritmus ML, který se v herním průmyslu používá pro klasifikaci i regresi dat. Jeho cílem je rozdělit data do kategorií (klasifikace) nebo předpovědět hodnoty (regrese) na základě vstupních dat. [19]

Princip fungování:

- **Hyperplocha:** Algoritmus vyhledá hyperplochu, která odděluje data do kategorií s co největším rozestupem. Tato hyperplocha slouží jako rozhodující hranice pro nová data.
- **Podpůrné vektory:** Algoritmus se zaměřuje na podpůrné vektory, což jsou body dat, které leží nejblíže hyperploše. Tyto body dat jsou klíčové pro definici hyperplochy a pro klasifikaci nových dat.
- **Maximalizace rozestupu:** Cílem SVM je maximalizovat rozestup mezi hyperplochou a nejbližšími body dat. To zajišťuje, že hyperplocha je co nejrobustnější a že dokáže správně klasifikovat i nová data, která se liší od trénovacích dat.

2.5.1.3 K-nearest neighbor

K-nearest neighbor (KNN) je klasifikační algoritmus ML, který se v herním průmyslu používá k rozpoznávání a kategorizaci dat. Jeho cílem je zařadit nová data do kategorie na základě jejich podobností s existujícími daty. [33]

Princip fungování:

- **Výpočet vzdálenosti:** Algoritmus vypočítá vzdálenost mezi novým datem a K nejbližšími daty v trénovací sadě. Vzdálenost se obvykle vypočítává pomocí Eukleidovy věty.
- **Hlasování:** Algoritmus určí kategorii nového data na základě kategorie nejbližších dat. KNN může používat různé strategie hlasování, jako je hlasování většiny nebo vážené průměrování.
- **Předpoklad podobnosti:** Algoritmus se opírá o předpoklad, že podobná data se vyskytují blízko sebe v prostoru. To znamená, že data, která jsou si navzájem nejblíže, patří pravděpodobně do stejné kategorie.

2.5.1.4 Random Forest

Random Forest je algoritmus ML, který se v herním průmyslu používá pro klasifikaci **a** regresi dat. Jedná se o soubor rozhodovacích stromů **(**DT**)**, které jsou kombinovány tak, aby dosáhly co nejlepšího výsledku. [20]

Princip fungování:

- **Náhodné rozdělení dat:** Algoritmus náhodně rozdělí data na mnoho podmnožin.
- **Vytvoření stromů rozhodování:** Pro každou podmnožinu dat se vytvoří DT. Stromy se budují tak, že se na každém uzlu vybere náhodný atribut a data se rozdělí na základě hodnoty tohoto atributu.
- **Hlasování:** Pro klasifikaci nového data se každý DT v Random Forestu použije k předpovědi kategorie. Kategorie s největším počtem hlasů se pak stane konečnou kategorií pro dané datum.
- **Agregace:** Algoritmus agreguje výsledky ze všech DT a vybere nejčastější kategorii pro dané datum.

2.5.1.5 Linear regression

Lineární regrese je statistická metoda používaná v herním průmyslu k modelování vztahů mezi závislou proměnnou a jednou nebo více nezávislými proměnnými. Cílem lineární regrese je najít nejlepší lineární závislost mezi těmito proměnnými, která dokáže co nejlépe vysvětlit pozorovaná data. [34]

Princip fungování:

- **Modelová hypotéza:** Lineární regrese předpokládá, že existuje lineární závislost mezi proměnnou (Y) a nezávislými proměnnými (X1, X2, …, Xn). Tato závislost se vyjadřuje rovnicí:

$$Y = \beta 0 + \beta 1 \times X1 + \beta 2 \times X2 + \cdots + \beta n \times Xn + \varepsilon$$

  Kde:

  - β0 je konstanta
  - β1, β2, …, βn jsou koeficienty regresních proměnných
  - ε je náhodná chyba

- **Metoda nejmenších čtverců:** Cílem lineární regrese je najít koeficienty β0, β1, β2, …, βn tak, aby se minimalizovala souhrnná chyba (suma čtverců reziduí). Toho se dosahuje pomocí metody nejmenších čtverců.
- **Interpretace výsledků:** Koeficienty β1, β2, …, βn ukazují, o kolik se závislá proměnná Y změní v průměru při změně příslušné nezávislé proměnné Xi o jednu jednotku, za předpokladu, že se ostatní nezávislé proměnné nezmění.

2.5.1.6 Logistic regression

Logistická regrese je statistická metoda používaná v herním průmyslu k modelování binárních dat, tj. dat, která mohou mít pouze dvě hodnoty (např. ano/ne, pravda/nepravda). Funguje podobně jako lineární regrese, ale s tím rozdílem, že nevytváří lineární závislost mezi proměnnými, ale modeluje pravděpodobnost výskytu určité události. [21]

Princip fungování:

- **Modelová hypotéza:** Logistická regrese předpokládá, že pravděpodobnost výskytu Y závisí na nezávislých proměnných X1, X2, …, Xn. Tato nezávislost se vyjadřuje rovnicí:

  $$P(Y = 1) = 1/(1 + \exp(-(\beta 0 + \beta 1 \times X1 + \beta 2 \times X2 + \cdots + \beta n \times Xn)))$$

  Kde:

  - P(Y = 1) je pravděpodobnost výskytu události Y
  - β0 je konstanta
  - β1, β2, …, βn jsou koeficienty regresních proměnných
- **Metoda maximální pravděpodobnosti:** Cílem logistické regrese je najít koeficienty β0, β1, β2, …, βn tak, aby se maximalizovala pravděpodobnost pozorovaných dat. Toho se dosahuje pomocí metody maximální pravděpodobnosti.
- **Interpretace výsledků:** Koeficienty β1, β2, …, βn ukazují, o kolik se změní logit (přirozený logaritmus poměru pravděpodobnosti) události Y při změně příslušné nezávislé proměnné Xi o jednu jednotku, za předpokladu, že se ostatní nezávislé proměnné nezmění.

2.5.2 Učení bez učitele

Na rozdíl od Učení s učitelem, které se spoléhá na označená tréninková data, Učení bez učitele prozkoumává daná data bez jakéhokoliv předchozího označení. Hledá v nich skryté

vzorce a podobnosti tzv. „naslepo“. Díky tomu, že si sám odvodí souvislosti bez lidského zásahu, se tento model nazývá Učení bez učitele.

Hlavní využití nachází v oblastech průzkumu dat, analýzy, rozpoznávání a dělení dat. Mezi nejznámější příklady patří algoritmy doporučování obsahu na platformách jako je YouTube a Google. [22]

Zjednodušeně řečeno:

- **Učení s učitelem:** Učí se z příkladů, kdy jsou data jasně označená (např. obrázky koček a psů).
- **Učení bez učitele:** Hledá skryté vzorce a souvislosti v neoznačených datech (např. doporučuje videa na základě sledovací historie).

Tři hlavní oblasti využití Učení bez učitele:

- **Clustering (Shlukování):** Je technika ML, která se zaměřuje na seskupování dat do tzv. klastrů. Tyto klastry sdružují datové body na základě jejich vzájemné podobnosti, ať už se jedná o vlastnosti, chování, nebo jiné relevantní atributy. Shlukování se používá v široké škále oblastí, od marketingu a biologie až po finance a informatiku. Hlavní výhodou je jeho univerzálnost, jelikož umožňuje zpracovávat neoznačená a neupravená data, která by pro jiné metody ML byla nevhodná. Existuje mnoho různých shlukovacích algoritmů, které se liší v přístupu a způsobu seskupování dat. Mezi nejběžnější patří: [22]
    - **Exkluzivní klastrování:** Každý datový bod patří do jednoho a pouze jednoho klastru
    - **Překrývající se klastrování:** Datový bod se může nacházet ve více klastrech současně.
    - **Hierarchické klastrování:** Shlukuje data do stromové struktury, kdy se menší klastry sdružují do větších.
    - **Pravděpodobnostní klastrování:** Přiřazuje datovým bodům pravděpodobnost, že patří do daného klastru.
- **Association rules (asociační analýzy):** Zabývá se hledáním vzájemných souvislostí mezi proměnnými v daném souboru dat. Tato metoda umožňuje odhalit skryté vazby a trendy, které by mohly být jinak přehlédnuty. V praxi se využívá v široké škále oblastí, od analýzy prodeje produktů v obchodě až po studium genetických mutací

a vlivu životního prostředí. Mezi nejčastěji používané algoritmy asociační analýzy patří: [22]

- **Apriori:** Tento algoritmus je považován za jeden z nejefektivnějších a nejrozšířenějších. Dokáže najít časté vzory v datech a odvodit z nich asociační pravidla.
- **Eclat:** Algoritmus Eclat se zaměřuje na nalezení vzájemných vazeb mezi skupinami položek. Je obzvlášť užitečný pro analýzu dat s velkým počtem proměnných.
- **FP-Growth:** Tento algoritmus se vyznačuje rychlou a efektivní implementací, která je vhodná pro zpracování rozsáhlých datových sad.

- **Dimensionality reduction (Redukce rozměrů):** Je technika, která se zaměřuje na zjednodušení datových sad. Jejím cílem je redukovat počet proměnných (tzv. dimenzí) v datech, a to s co nejmenším dopadem na výsledky analýzy. Toho je dosažené identifikací a odstraněním redundantních nebo irelevantních informací, čímž se zjednodušuje a zefektivňuje práce s daty. Hlavní využití nalezneme v oblastech, kde je nutné zpracovávat rozsáhlé datové sady, jako je ML, analýza dat, nebo bioinformatika. Mezi nejčastěji používané metody patří: [22]
  - **Analýza hlavních komponent (PCA):** Tato metoda transformuje data do nového souřadnicového systému tak, aby první osa představovala směr s největší variabilitou dat. Následující osy pak zachycují směry s menší variabilitou.
  - **Dekompozice singulární hodnoty (SVD):** SVD rozkládá datovou matici na tři matice, z nichž jedna slouží k redukci dimenzionality výběrem singulárních hodnot s největšími absolutními hodnotami.
  - **Autoenkóder:** Tato metoda neuronové sítě se učí kódovat a dekódovat data. Během kódování se síť zaměřuje na zachycení nejdůležitějších informací v datech s redukovaným počtem dimenzí.

2.5.3 Zpětnovazební učení

Zpětnovazební učení (RL) se odlišuje od klasického ML tím, že se agent (např. robot, algoritmus) učí sám na základě pokusů a omylů, bez nutnosti předchozího označení dat. Agent se pohybuje v daném prostředí a provádí akce, za které dostává odměny nebo tresty. Na

základě těchto zpětných vazeb se agent snaží optimalizovat své chování a dosáhnout co největší odměny. [23]

Hlavní principy:

- **Agent**: Ten, kdo se učí a provádí akce v prostředí
- **Prostředí**: Poskytuje agentovi informace o jeho stavu a reaguje na jeho akce
- **Akce**: Možnosti, které agent může provést v daném prostředí
- **Odměna**: Pozitivní signál, který agent získá za žádoucí chování
- **Trest**: Negativní signál, který agent získá za nežádoucí chování

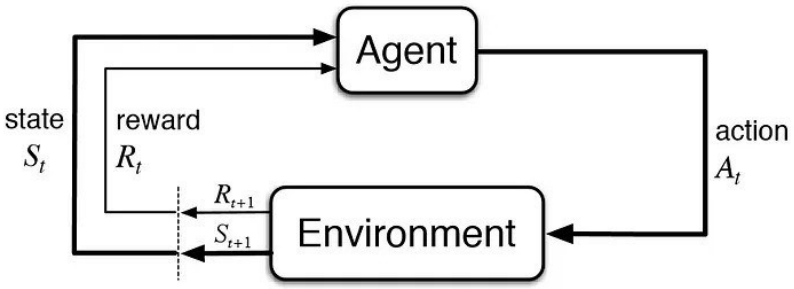
Využití RL:

- **Optimalizace robotů:** RL umožňuje robotům učit se efektivněji pohybovat a plnit úkoly v neznámých prostředích.
- **Vytváření herních AI:** RL se používá k vývoji herních protivníků, kteří se dokáží učit a přizpůsobovat hernímu stylu hráče.
- **Finanční modelování:** RL se může použít k predikci tržních trendů a optimalizaci investičních strategií.
- **Lékařská diagnostika:** RL algoritmy mohou analyzovat data pacientů a napomáhat lékařům s diagnostikou onemocnění.

V RL se agenti snaží nalézt optimální strategii pro dosažení co největší v daném prostředí. To však představuje složitou výzvu, jelikož je nutné vyvážit dva protichůdné cíle:

- **Průzkum (Exploration):** Agent zkoumá nové stavy a akce, aby objevil potenciálně výnosnější strategie. To může znamenat riskování a dočasné snižování odměny.
- **Využívání (Exploitation):** Agent se zaměřuje na akce, o kterých ví, že vedou k vysoké odměně. To může znamenat uvíznutí v lokálním optimu a nalezení lepších strategií v dlouhodobém horizontu.

Najít optimální strategii znamená najít rovnováhu mezi těmito dvěma cíli. Pokud agent příliš zkoumá, riskuje, že ztratí čas a zdroje bez dosažení významného pokroku. Naopak, pokud se zaměří pouze na maximalizaci odměny, může si nechat ujít příležitost k nalezení lepších strategií v dlouhodobém horizontu. [23]



Obr. 6 Zpětnovazební učení, Zdroj: https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292

Při vývoji algoritmů RL se používá několik klíčových metod, které umožňují agentům učit se z odměn a trestů v daném prostředí. Mezi dva nejvýznamnější příklady patří:

- **Markovský rozhodovací proces (MDP):** MDP je matematický model, který popisuje dynamiku prostředí, ve kterém se agent pohybuje. Cílem agenta MDP je najít optimální strategii, která mu umožní maximalizovat kumulativní odměnu v průběhu času. Skládá se ze čtyř hlavních komponent: [23]
  - **Stavy:** Množina všech akcí, které agent může v daném stavu provést.
  - **Přechodové funkce:** Funkce, která udává pravděpodobnost přechodu z jednoho stavu do druhého po provedení dané akce.
  - **Odměnová funkce:** Funkce, která přiřazuje odměnu za dosažení daného stavu.
- **Q-learning:** Algoritmus RL, který se používá k nalezení optimální strategie v MDP. Funguje na principu učení hodnoty jednotlivých akcí v daných stavech. Q-learning je iterativní algoritmus, který se učí a zlepšuje svou strategii s každým krokem. Skládá se z následujících kroků: [23]
  - **Inicializace Q-tabulky**: Vytvoří se tabulka, která ukládá odhadované hodnoty Q pro všechny stavy a akce.
  - **Výběr akce:** Agent v aktuálním stavu vybere akci na základě aktuální Q-tabulky. Existuje několik metod výběrů akcí, například:
    - **Epsilon-greedy:** S pravděpodobností ε se vybere náhodná akce a s pravděpodobností (1–ε) se vybere akce s nejvyšší hodnotou Q v daném stavu.
    - **Boltzmann:** Pravděpodobnost výběru akce je úměrná exponenciální funkcí její hodnoty Q.

- **Provedení akce a pozorování**: Agent provede vybranou akci a pozoruje nový stav a odměnu, kterou za ni obdrží.
- **Aktualizace Q-tabulky:** Hodnota Q pro aktuální stav a akci se aktualizuje podle vzorce:

$$Q(s,a) \leftarrow Q(s,a) + a(R(s) + yQ(s',a') - Q(s,a))$$

Kde:

- **a** je koeficient učení, který určuje, jak moc se nová informace ovlivní stávající hodnotou Q.
- **R(s)** je odměna získaná v aktuálním stavu s.
- **γ** je diskontní faktor, který určuje, jak moc se budou zohledňovat budoucí odměny.
- **s'** je nový stav po provedení akce a.
- **a'** je akce vybraná v novém stavu s'.

- **Přechod do dalšího kroku:** Agent se přesune do nového stavu a proces se opakuje od kroku 2 (Výběr akce).

Výhody Q-learning:

- Q-learning je relativně jednoduchý algoritmus, který se snadno implementuje.
- Je schopen se učit v MDP s velkým stavovým prostorem a diskrétními i spojitými akcemi.
- Nevyžaduje znalost modelu prostředí.

Nevýhody Q-learning:

- Q-learning se může učit pomalu, zvláště v MDP s velkým stavovým prostorem.
- Algoritmus může být citlivý na výběr koeficientu učení a diskontního faktoru.
- Q-learning se může uvíznout v lokálních optimech.

Příklady použití Q-learning:

- **Robotika:** Q-learning se může používat k učení robotů, jak plnit úkoly, jako je navigace v prostředí nebo manipulace s předměty.

- **Financování:** Q-learning se může používat k učení se optimálním investičním strategiím.
- **Hraní her:** Q-learning se může používat k učení agentů, jak hrát hry, jako je Šach nebo Go.

2.5.3.1 Deep Q-Networks

Deep Q-Networks (DQN) je algoritmus RL, který se odlišuje od klasických RL metod tím, že pro estimaci Q-hodnot (hodnot akcí v daných stavech) využívá neuronové sítě. Díky tomu je schopen zvládat komplexnější úlohy v prostředí s větším počtem dimenzí. [35]

I když DQN představuje významný krok vpřed v oblasti RL, je nutné zmínit i jeho omezení. Algoritmus je primárně určen pro prostředí s nízkým rozměrem stavového prostoru. [35] V případě komplexních prostředí s vysokým rozměrem může být učení neuronové sítě obtížné a časově náročné. [35]

Princip fungování:

- **Iniciace:** Vytvoří se neuronová síť s počtem vstupů odpovídajícím počtu dimenzí stavu a počtem výstupů odpovídajícím počtu akcí. Síť je inicializována náhodnými váhami.
- **Interakce s prostředím:** Agent provádí akci v aktuálním stavu a pozoruje nový stav a odměnu.
- **Aktualizace neuronové sítě:** Na základě aktuální odměny a odhadovaných Q-hodnot v novém stavu se aktualizují váhy neuronové sítě.
- **Výběr další akce:** Agent volí další akci na základě predikovaných Q-hodnot z neuronové sítě.

2.5.3.2 Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient (DDPQ) je další algoritmus RL, který se odlišuje od DQN tím, že využívá tzv. Actor-Critic architekturu. Tato architektura umožňuje DDPQ zvládat úlohy ve vysokorozměrném prostředí s kontinuálním akčním prostorem. [36]

Princip fungování:

- **Inicializace:** Vytvoří se dvě neuronové sítě: Actor, která generuje akce na základě stavu, a Critic, která hodnotí Q-hodnoty (hodnoty akcí) v daných stavech.

- **Interakce s prostředím:** Agent provádí akci vygenerovanou Actor sítí a pozoruje nový stav a odměnu.
- **Aktualizace Critic sítě:** Na základě aktuální odměny a odhadovaných Q-hodnot v novém stavu se aktualizují váhy Critic sítě.
- **Aktualizace Actor sítě:** Váhy Actor sítě se aktualizují tak, aby se maximalizovala očekávaná Q-hodnota pro daný stav.
- **Výběr další akce:** Agent volí další akci na základě predikovaných Q-hodnot z Critic sítě.

2.5.4 Neuronové sítě

Neuronové sítě (ANN) představují typ ML, který se inspiruje strukturou a funkcí lidského mozku. Skládají se z propojených uzlů, tzv. neuronů, uspořádaných do vrstev: vstupní, skryté a výstupní. Každý neuron je propojen s dalšími neurony v následující vrstvě a tato propojení nesou specifické váhy a prahy aktivace. Informace prochází sítí směrem vpřed, od vstupní vrstvy ke skrytým vrstvám a dále do výstupní vrstvy. Vstupní vrstva přijímá data, která jsou následně zpracována a transformována vrstvami skrytých neuronů. Výstupní vrstva pak generuje finální výsledek na základě aktivace neuronů v předchozích vrstvách. Aktivace neuronu nastává, pokud součet jeho vstupů vynásobených příslušnými váhami překročí práh aktivace. V takovém případě neuron „vystřelí“ a pošle svůj signál do neuronu v další vrstvě. [24][25]

Podobně jako Učení s učitelem, i ANN se spoléhají na trénovací data pro optimalizaci své přesnosti. Během trénování se síť učí na vzorcích dat, přičemž se upravují váhy propojení mezi neurony tak, aby minimalizovala chybu mezi předpovězeným a požadovaným vstupem. [24][25]

Známé využití ANN:

- **Vyhledávací algoritmus Google:** ANN hrají klíčovou roli v algoritmu Google pro vyhledávání, kde pomáhají s hodnocením relevance webových stránek a jejich zobrazováním v relevantních výsledcích vyhledávání.
- **Systémy autonomního řízení:** ANN se používají v systémech autonomního řízení pro rozpoznávání objektů v okolí vozu, predikci trajektorie pohybu a následné rozhodování o řízení.

- **Lékařská diagnostika:** ANN se zkoumají i v oblasti lékařské diagnostiky, kde se učí na datech pacientů a napomáhají lékařům s diagnostikou onemocnění.

2.5.4.1 Feedforward neural networks

Feedforward neural networks (FNN) představují jeden ze základních typů ANN, kde uzly (neurony) jsou uspořádány do vrstev a informace se šíří pouze jedním směrem, od vstupní vrstvy přes skryté vrstvy až k výstupní vrstvě. [24][25]

Princip fungování:

- **Vstupní vrstva:** Přijímá data, která se následně zpracovávají a transformují v dalších vrstvách.
- **Skryté vrstvy:** Provádějí hlubší zpracování dat a extrahují abstraktní rysy. Počet skrytých vrstev a počet neuronů v nich ovlivňuje komplexnost modelu a jeho schopnost učit se složitým závislostem v datech.
- **Výstupní vrstva:** Generuje finální výsledek na základě aktivace neuronů v předchozích vrstvách.

2.5.4.2 Convolutional neural networks

Convolutional neural networks (CNN) představují typ ANN inspirovaný principy z lineární algebry a násobení matic. Na rozdíl od FNN, kde se informace šíří pouze jedním směrem, CNN využívají lokální operace pro zpracování dat v mřížkové struktuře (např. obrázky). To je činí ideálními pro úkoly jako je rozpoznávání objektů v obrazech, segmentace obrazu a analýza videa. [24][25]

Princip fungování

- **Konvoluční jádra:** Základní stavební kameny CNN. Skládají se z malé mřížky vah, které se posouvají po vstupním obrazu a provádí s ním bodové násobení a součet.
- **Aktivační funkce:** Aplikuje se na výsledek konvoluce a zavádí nelinearitu do modelu.
- **Pooling:** Slouží ke snížení rozměrů dat a abstrakci lokálních rysů. Existuje mnoho poolingových operací, jako je maximální pooling a průměrný pooling.
- **Skryté vrstvy:** CNN se skládají z více konvolučních a poolingových vrstev, které se střídají. Každá vrstva extrahuje abstraktnější rysy z dat.

- **Výstupní vrstva:** Generuje finální výsledek, například klasifikaci objektu v obrázků.

2.5.4.3 Recurrent neural networks

Recurrent neural networks (RNN) se odlišují od FNN a CNN jedinečnou vlastností: zpětnovazebními smyčkami. Tyto smyčky umožňují RNN uchovávat si informace o minulých vstupech a používat je pro zpracování aktuálních dat. Díky tomuto mechanismu paměti se RNN stávají ideálními pro úkoly, kde je potřeba zohlednit časovou posloupnost dat, jako je předpovídaní budoucích trendů, strojový překlad a generování textu. [24][25]

Princip fungování:

- **Skrytá vrstva s pamětí:** V srdci RNN se nachází skrytá vrstva s pamětí, která si uchovává informace o minulých vstupech. Tato vrstva se nazývá „rekurzivní vrstva“ a je klíčová pro fungování RNN.
- **Váhy a aktivační funkce:** Skrytá vrstva s pamětí je propojena s aktuálním vstupem a předchozím stavem paměti pomocí vah. Výsledek se pak zpracuje aktivační funkcí, která zavádí nelinearitu do modelu.
- **Výstupní vrstva:** Generuje finální výsledek na základě stavu paměti a aktuálního vstupu.

2.5.4.4 NEAT algorithm

NEAT (NeuroEvolution of Augmenting Topologies) je algoritmus pro neuroevoluci, což je oblast AI, která kombinuje neuronové sítě s evolučními algoritmy. NEAT se odlišuje od jiných neuroevolučních algoritmů v tom, že umožňuje automatické změny topologie neuronové sítě během procesu evoluce. To znamená, že algoritmus NEAT může nejen upravovat váhy spojení mezi neurony, ale také přidávat, odebírat a měnit neurony a spojení v síti. [37]

Základní principy NEAT:

- **Populační přístup:** NEAT pracuje s populací neuronových sítí, které se v průběhu času vyvíjejí.
- **Genetické variace:** Nové sítě vznikají mutacemi a křížením existujících sítí v populaci.
- **Selekce na základě fitness:** Sítě s lepším výkonem (vyšší fitness) se s větší pravděpodobností stanou rodiči pro další generaci.

- **Dynamická topologie:** NEAT umožňuje měnit svou topologii (počet neuronů a spojení) během evoluce.

Hlavní vlastnosti NEAT:

- **Vyhýbání se konkurenčním konvencím:** NEAT používá historické značení nových strukturálních prvků (neuronů, spojení) k tomu, aby zabránil vzniku sítí s identickým chováním, ale s odlišnou genotypickou reprezentací. To umožňuje efektivnější křížení a zabraňuje ztrátě genetické informace. [37]
- **Speciace a sdílení fitness:** NEAT rozděluje populaci na druhy na základě podobnosti genotypů. Křížení a mutace probíhají s větší pravděpodobností uvnitř druhů, čímž se podporuje konvergence a zabraňuje se vzniku nekompatibilních potomků. [37]
- **Zdokonalování:** NEAT začíná s jednoduchými sítěmi a postupně je zdokonaluje mutacemi, které přidávají nové neurony a spojení. To umožňuje sítím se přizpůsobit složitým problémům bez nutnosti manuálního nastavování topologie. [37]

Výhody NEAT:

- **Automatické hledání optimální topologie:** NEAT umožňuje neuronovým sítím najít optimální topologii pro daný problém bez nutnosti ručního ladění.
- **Robustnost vůči lokálním minimům:** NEAT je méně náchylný k uvíznutí v lokálních minimech než tradiční metody optimalizace neuronových sítí.
- **Schopnost řešit složité problémy:** NEAT byl úspěšně použit k řešení široké škály složitých problémů, včetně robotiky, her a predikce časových řad.

Nevýhody NEAT:

- **Vysoké výpočetní nároky:** NEAT může být výpočetně náročný, zejména u velkých populací a složitých problémů.
- **Obtížnost nastavení parametrů:** NEAT má mnoho parametrů, které je nutné nastavit pro dosažení optimálního výkonu. To může být pro začátečníky obtížné.

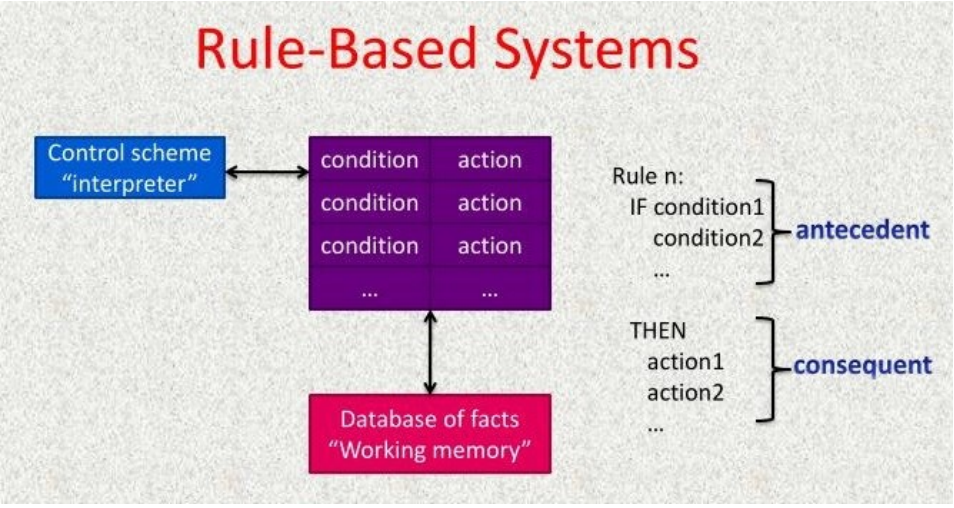
2.6 Rule-based systems

Systémy založené na pravidlech (RBS) představují základní formu AI, která spočívá v definování předpřipravených pravidel pro chování nehráčských postav (NPC) ve hře. Tato metoda umožňuje dosahovat jasně definovaných, předvídatelných a rychlých rozhodnutí NPC.

Implementace RBS probíhá buď pomocí DT, které na základě zadaných podmínek vedou k určitému výsledku, nebo jednoduše pomocí podmínek a jejich následků (if-then statements). V herním vývoji se RBS běžně používá jak pro tvorbu samotných NPC, tak i pro simulaci jejich chování v herním světě. [26]

RBS se skládají ze tří klíčových komponent:

- **Soubor faktů:** Jedná se o databázi informací, která slouží jako zdroj dat pro RBS. Tyto informace mohou zahrnovat jak statická data (např. vlastnosti NPC), tak i dynamická data (např. aktuální pozice hráče). [26]
- **Soubor pravidel:** Tato sada definuje logiku chování NPC. Pravidla obvykle sestávají z podmínek a následků, které určují, jak NPC zareaguje na danou situaci. Podmínky specifikují, kdy se pravidlo má aplikovat, zatímco následky definují, jakou akci má NPC provést. [26]
- **Ukončovací kritérium:** Toto kritérium určuje, kdy se cyklus zpracování RBS ukončí. Ukončovací kritérium může být založeno na splnění určitého cíle, dosažení časového limitu nebo na jiných faktorech. [26]



Obr. 7 Rule-based systems, Zdroj: https://www.slideserve.com/aricin/cpts-440-1324085

3 MOŽNOSTI A LIMITY UMĚLÉ INTELIGENCE V MOBILNÍCH HRÁCH

3.1 Možnosti AI v mobilních hrách

Vzhledem k inherentním limitům hardwarových specifikací mobilních zařízení je nezbytné při navrhování her pro tuto platformu zohledňovat jejich plynulý chod a optimalizaci. Vývojáři her tak v rostoucí míře začleňují AI do různých fází vývojového procesu, ať už s cílem optimalizace herního výkonu, zefektivnění práce, nebo pro dosažení pohlcujícího herního zážitku. Zde je několik příkladů použití, kdy AI pomáhá ve vývoji mobilních her: [27]

Optimalizace grafiky a herního výkonu:

Sofistikované AI algoritmy dokáží analyzovat herní scény a idenfitikovat oblasti, které je možné zjednodušit bez kompromisů v oblasti vizuální kvality. To umožňuje optimalizaci herních grafik a dosažení plynulého herního chodu i na zařízeních s nižším výkonem. [27]

Generování herního obsahu:

AI nástroje pomáhají automatické tvorbě herních prvků, jako jsou herní prostředí, postavy, nebo herní mechanismy. To vývojářům umožňuje ušetřit čas a zaměřit se na kreativnější aspekty herního designu. [27]

Personalizace herního zážitku:

Implementace AI umožňuje analyzovat herní chování uživatelů a na základě získaných dat personalizovat herní prostředí dle individuálních preferencí. To zahrnuje dynamické úpravy obtížností, tempa hry, nebo výběru herního obsahu. [27]

Automatizované testování a ladění:

AI asistenti usnadňují proces testování herního softwaru a automaticky identifikují chyby a problémy. To urychluje ladění a zajišťuje stabilitu a plynulost herního prostředí. [27]

Tvorba chytrých herních protivníků:

Využití AI umožňuje implementovat sofistikované herní protivníky, kteří dokáží reagovat na herní styl uživatele a adaptivně se mu přizpůsobovat, čímž se zvyšuje dynamika a atraktivita herního zážitku. [27]

Integrace AI do vývoje mobilních her otevírá široké spektrum možností pro inovace a zkvalitnění herního prostředí. S rostoucím výkonem mobilních zařízení a kontinuálním vývojem

v oblasti AI můžeme očekávat, že se tato technologie stane nepostradatelným nástrojem pro tvůrce her a posune mobilní hraní na novou úroveň.

3.2 Limity v mobilních hrách s AI

AI v mobilních hrách má velký potenciál, ale stále existují určité limity, které brání jejímu plnému využití. Mezi nejvýznamnější limity patří: [27]

Technické výzvy:

- **Výkonová omezení:** Mobilní zařízení obecně disponují menším výpočetním výkonem a pamětí v porovnání s PC a konzolemi. To může vést k problémům s plynulým chodem náročných AI modelů na mobilních zařízení. [27]
- **Optimalizace:** Pro dosažení optimálního výkonu a výdrže baterie je nutná pečlivá optimalizace AI systémů pro mobilní platformy. To zahrnuje efektivní využití dostupných hardwarových zdrojů a minimalizaci energetické náročnosti. [27]
- **Výkon versus výdrž baterie:** V praxi je nutné najít kompromis mezi dosažením požadovaného výkonu AI modelů a udržením adekvátní výdrže baterie mobilních zařízení. [27]

Omezení zdrojů:

- **Vysoké nároky na zdroje:** Algoritmy AI v mobilních hrách vyžadují značné množství výpočetních zdrojů a dat pro dosažení optimálního výkonu. To může vést k problémům s plynulým chodem hry, pokud nejsou zdroje spravovaný efektivně. [27]
- **Vyvažování nároků:** Mobilní hry musí vyvážit nároky AI s ostatními herními komponenty, jako je grafika, fyzika a síťové připojení. Pokud AI spotřebuje příliš mnoho zdrojů, může to vést ke zpomalení hry, snížení snímkové frekvence a dalším problémům s výkonem. [27]

4 ETICKÉ ASPEKTY UMĚLÉ INTELIGENCE V HERNÍM PROSTŘEDÍ

Ochrana osobních údajů a bezpečnost:

- **Sběr dat a obavy o soukromí:** Implementace AI v hrách často zahrnuje sběr a analýzu uživatelských dat pro optimalizaci herního zážitku. To vyvolává obavy o soukromí a bezpečnost dat hráčů. [27]
- **Dodržování předpisů:** Vývojáři her s AI musí dodržovat platné předpisy o ochraně osobních údajů, jako je GDPR v Evropě a CCPA v Kalifornii. To zahrnuje získání souhlasu uživatelů se sběrem dat, transparentnost ohledně shromažďovaných dat a jejich použití a implementaci adekvátních bezpečnostních opatření. [27]
- **Robustní bezpečnostní opatření:** Pro ochranu uživatelských dat před neoprávněným přístupem, zneužitím nebo ztrátou je nutné zavést robustní bezpečnostní opatření. To zahrnuje šifrování dat, ochranu před kybernetickými útoky a implementaci jasných procesů pro správu incidentů. [27]

Předpojatosti a spravedlnost algoritmů:

- **Neúmyslné šíření předsudků:** Algoritmy AI v hrách se učí z dat, která jim jsou poskytnuta. Pokud tato data obsahují předsudky, algoritmy je mohou neúmyslně šířit a promítat do herního světa. To může vést k nefér nebo diskriminačním výsledkům pro hráče, kteří jsou součástí znevýhodněných skupin. [27]
- **Stereotypizace:** Algoritmy AI v hrách s genderovou tématikou se mohou učit ze stereotypních zobrazení mužů a žen, které pak promítají do hry. To může vést k tomu, že jsou určité herní postavy nebo herní mechaniky nevhodné pro specifické pohlaví. [27]
- **Diskriminace:** Algoritmy AI v hrách s online multiplayerem se mohou učit z dat, která obsahují rasové nebo etnické předsudky. To může vést k tomu, že jsou někteří hráči nespravedlivě penalizováni nebo vyloučeni z herních komunit. [27]

Manipulace a kontrola:

- **Návykovost:** AI může být použita k personalizaci herního zážitku pro každého hráče tak, aby byl co nejvíce návykový. To může vést k tomu, že hráči budou trávit hraním her více času, než by si přáli, a že budou mít potíže s přestáním hrát. [27]

- **Psychické manipulace:** AI může být použita k manipulaci s emocemi hráče a k jejich ovlivňování. Toho lze dosáhnout například pomocí AI k vytváření herních světů, které jsou plné úzkostí a stresu, nebo k zobrazování cílených reklam, které hrají na lidské slabosti. [27]
- **Sociální inženýrství:** AI může být použita k podvádění hráčů a k jejich oklamání, aby dělali to, co vývojáři her chtějí. Toho lze dosáhnout například pomocí AI k vytváření falešných hráčů, kteří s hráči interagují, nebo k šíření dezinformací v herním světě. [27]

II. PRAKTICKÁ ČÁST

5 FLAPPY BIRD

Tato kapitola se zabývá mobilní hrou Flappy Bird. Kapitola začíná popisem samotné hry a principem jejího fungování. Dále se zaměřuje na implementaci algoritmu NEAT a na vytvoření simulovaného prostředí hry Flappy Bird pomocí Pythonu.

5.1 Popis a herní principy

Flappy Bird je mobilní hra z roku 2013, která si získala popularitu, ale i pověst frustrující hratelností.

Základní myšlenka:

- Hra je typu „side-scroller“, kde vidíte herní svět z boku a postava se pohybuje zleva doprava. Úkolem je ovládat malého ptáčka, který neustále klesá k zemi.
- Jediný způsob, jak ptáčka udržet ve vzduchu, je klepat na obrazovku. Čím déle držíte prst na displeji, tím výše pták vylétne.
- Hlavní překážkou ve hře jsou řady zelených pevných trubek s mezerami mezi nimi. Cílem je manévrovat ptáčkem tak, aby prolétl mezi těmito trubkami, aniž by se jich dotkl.

Herní principy:

- Čím dál do hry se dostanete, tím rychleji se trubky posouvají a mezery mezi nimi se zmenšují. To vyžaduje přesné načasování a rychlé reakce.
- Hra je velice jednoduchá na pochopení, ale těžká na zvládnutí. To je jedním z hlavních důvodů její popularity.
- Díky vysoké obtížnosti a náhodnému uspořádání trubek je hra nekonečně znovuhratelná.

5.2 Python

Python je vysokoúrovňový programovací jazyk, to znamená, že je relativně snadno čitelný a psát se v něm dá intuitivním způsobem. Navíc je všestranný, takže se hodí pro širokou škálu úkolů, od vývoje webových stránek a analýzy dat až po strojové učení a umělou inteligenci. [28]

Mezi hlavní vlastnosti Pythonu patří:

- **Čitelnost:** Kód pythonu je snadno pochopitelný i pro začátečníky, díky čemuž je ideální pro výuku programování.
- **Všestrannost:** Python lze použít pro širokou škálu úkolů, od webového vývoje až po vědecké výpočty.
- **Velká komunita:** Python má velkou a aktivní komunitu vývojářů, což znamená, že je snadné najít pomoc a podporu online.
- **Mnoho knihoven:** Existuje mnoho knihoven pro Python, které poskytují funkce pro různé úkoly, jako je strojové učení, analýza dat a vývoj webových stránek.

Využití Pythonu:

- **Webový vývoj:** Python se běžně používá pro vývoj webových stránek a webových aplikací pomocí frameworků jako Django a Flask.
- **Vědecké výpočty:** Python je populární volbou pro vědecké výpočty a analýzu dat díky knihovnám jako NumPy, SciPy a Pandas.
- **Strojové učení:** Python je jedním z předních jazyků pro strojové učení a umělou inteligenci díky knihovnám jako TensorFlow a PyTorch.
- **Automatizace:** Python se dá použít k automatizaci úkolů v různých oblastech, jako je správa systému, testování softwaru a zpracování dat.
- **Vývoj skriptů:** Python se často používá pro vývoj skriptů pro automatizaci úkolů v jiných aplikacích.

5.2.1 Pygame

Pygame je knihovna pro Python, která se používá k vývoji multimediálních aplikací, zejména videoher. Můžete si ji představit jako sadu nástrojů, které usnadňují programování poutavých her v Pythonu.

Pygame je bezplatná a open-source knihovna, což znamená, že je zdarma k použití a jeho komunitně podporovaný kód může být upraven a vylepšován. Díky tomu je Pygame skvělý nástroj pro začátečníky, kteří chtějí začít s vývojem her, aniž by se museli zapojovat do složitostí vývojářských nástrojů zaměřených na hry. Pygame je také multiplatformní, což znamená, že hry vytvořené pomocí Pygame lze hrát na Windows, macOS a Linux systémech, aniž by bylo nutné je znovu programovat pro každou platformu. [29][30]

Pygame poskytuje funkce pro různé aspekty vývoje her, včetně:

- **Zobrazování grafiky:** Můžete vytvářet jednoduché tvary, animovat pohyblivé objekty, a dokonce i načítat a zobrazovat obrázky a text na herní obrazovce.
- **Přehrávání zvuku:** Můžete přidávat zvukové efekty a hudbu na pozadí, aby bylo hraní ještě poutavější.
- **Zpracování vstupu:** Pygame umožňuje reagovat na vstup z klávesnice, myši a gamepadu. To je zásadní pro ovládání hry hráčem.

5.3 Implementace

Implementace herního prostředí Flappy Bird a NEAT algoritmu proběhla pomocí YouTube tutoriálu, který jsem doplnil o přidání těžší obtížnosti. [31]

Než začneme se samotnou implementací musíme si nainstalovat dvě knihovny, které budeme používat: Pygame a NEAT Python. To uděláme tak, že si vytvoříme nový projekt a otevřeme konzoli, do které vložíme následující příkazy: pip install pygame a pip install neat-python. Další věc, co musíme udělat je stáhnout si obrázky, které později použijeme pro našeho ptáčka, trubky, pozadí a zemi. Jakmile máme nainstalované obě knihovny a stáhnuté obrázky můžeme se pustit do samotné implementace kódu.

5.3.1 Moduly

Projekt využívá následující moduly:

- *pygame:* Pro vytvoření prostředí hry Flappy Bird
- *neat:* Pro implementaci metody NEAT
- *os:* Pro interakci s operačním systémem
- *random:* Pro generaci náhodných čísel
- *visualize:* Python skript, vytvořený pro vizualizaci výsledků v grafu

Tyto moduly jsou pomocí příkazu *import* načteny do programu pro přístup k jejich obsahu (viz. Obr. 8).

```
1  import pygame
2  import neat
3  import os
4  import random
5  import visualize
```

Obr. 8 Použité moduly pro tvorbu programu

5.3.2 Globální proměnné

Samotný program začíná nastavením globálních proměnných jako je font, obrázky, šířka a výška okna hry Flappy Bird, výška země, přiřazení obrázků a naposled nastavení proměnné *gen* na nulu pro nultou generaci. Pomocí *pygame.font.init()* inicializujeme modul pro font a pomocí *pygame.font.SysFont(„comicsans“, 50)* nastavíme, jaký font bychom chtěli používat a jeho velikost, v tomto případě je to font Comic Sans a jeho velikost je 50 px (pixelů). Dále si nastavíme velikost okna na 600 px a šířku okna na 800 px pomocí proměnných *WIN_WIDTH* a *WIN_HEIGHT*, které použijeme pro inicializaci okna pomocí *pygame.display.set_mode(WIN_WIDTH, WIN_HEIGHT).* Následovně si načteme obrázky, které jsme si připravili. Obrázky pro ptáčka si dáme do listu, jelikož máme tři obrázky, které nám budou simulovat animaci ptáčka. Obrázky si uložíme do proměnných pomocí *pygame.transform.scale2x* a *pygame.image.load*, které uloží obrázky do proměnných a zvětší jejich velikost dvakrát.

```
WIN_WIDTH = 600
WIN_HEIGHT = 800
STAT_FONT = pygame.font.SysFont( name: "comicsans",  size: 50)
pygame.display.set_caption("Flappy Bird")

WIN = pygame.display.set_mode((WIN_WIDTH, WIN_HEIGHT))

BIRD_IMGS = [pygame.transform.scale2x(pygame.image.load(os.path.join("imgs", "bird1.png"))),
             pygame.transform.scale2x(pygame.image.load(os.path.join("imgs", "bird2.png"))),
             pygame.transform.scale2x(pygame.image.load(os.path.join("imgs", "bird3.png")))]
PIPE_IMG = pygame.transform.scale2x(pygame.image.load(os.path.join("imgs", "pipe.png")).convert_alpha())
BASE_IMG = pygame.transform.scale2x(pygame.image.load(os.path.join("imgs", "base.png")).convert_alpha())
BG_IMG = pygame.transform.scale(pygame.image.load(os.path.join("imgs", "bg.png")).convert_alpha(),  size: (600, 900))
```

Obr. 9 Definice globálních proměnných

5.3.3 Vytvoření tříd a jejich metod

5.3.3.1 Třída Bird

Jako první třídu si vytvoříme třídu Bird, která reprezentuje našeho ptáčka a má metody *jump, move, draw a get_mask.* ***Def __init__(selfl, x, y)*** je metoda, která nám inicializuje našeho ptáčka. Vstupem jsou tři proměnné *self*, *x* a *y*, kde *x a y* reprezentují horizontální (osa x) a vertikální (osa y) pozici ptáčka a *self*, která představuje objekt třídy Bird. Definujeme si proměnné *self.x* s hodnotou *x*, *self.y* s hodnotou *y*, *self.vel* (rychlost ptáčka) s hodnotou 0, *self.tilt* (náklon ptáčka) s hodnotou 0, *self.tick_count* (slouží, abychom věděli, kdy ptáček naposled mávnul křídly) s hodnotou 0, *self.height* (výška ptáčka) s hodnotou *self.y,*

self.img_count (slouží, abychom věděli, jaký obrázek použít pro animaci ptáčka) s hodnotou 0, *self.img* s hodnotou *self.IMGS[0]* (použije při inicializaci první obrázek z listu IMGS).

```
class Bird:
    MAX_ROTATION = 25
    IMGS = BIRD_IMGS
    ROT_VEL = 20
    ANIMATION_TIME = 5

    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.tilt = 0  # degrees to tilt
        self.tick_count = 0
        self.vel = 0
        self.height = self.y
        self.img_count = 0
        self.img = self.IMGS[0]
```

Obr. 10 Metoda *__init__* ve třídě Bird

Dále máme metodu ***def jump(self)***, kde nastavíme rychlost našeho ptáčka *self.vel* na -10,5. Negativní rychlost dáváme, proto aby se ptáček pohyboval směrem nahoru, pokud bychom dali rychlost pozitivní, tak se ptáček pohybuje směrem dolů. Nastavíme *self.tick_count* na hodnotu 0 a *self.height* (slouží, abychom věděli, z jaké polohy ptáček skočil) na *self.y*.

Metoda ***def move(self)*** je metoda pomocí, které se ptáček pohybuje. Jako první přidáme *self.tick_count* hodnotu 1 pokaždé, co se ptáček pohne, čímž budeme sledovat, kdy naposled ptáček skočil. Dále si musíme vypočítat zrychlení našeho ptáčka, které nám řekne, jestli se pohybujeme nahoru nebo dolů. Následovně musíme nastavit konečnou rychlost, to znamená, pokud zrychlení dosáhne této hodnoty tak už se ptáček dál nezrychluje.

```
# downward acceleration
displacement = self.vel * self.tick_count + 1.5 * self.tick_count ** 2

# terminal velocity
if displacement >= 16:
    displacement = (displacement / abs(displacement)) * 16
```

Obr. 11 Výpočet zrychlení a konečné rychlosti v metodě *move* ve třídě Bird

Poslední věc, co uděláme v této metodě je nastavení sklonu ptáčka. To uděláme tak, že si prvně zjistíme, jestli se ptáček pohybuje směrem nahoru nebo pokud je pozice našeho ptáčka o 50 px vyšší od posledního skoku ptáčka, což znamená, že ptáček se pořád pohybuje směrem nahoru. Pokud jsou tyto podmínky pravdivé nastavíme *self.tilt* na hodnotu *self.MAX_ROTATION*. Pokud pravdivé nejsou, tak od *self.tilt* odečítáme hodnotu *self.ROT_VEL*.

```
# if moving upwards tilt up
if displacement < 0 or self.y < self.height + 50:
    if self.tilt < self.MAX_ROTATION:
        self.tilt = self.MAX_ROTATION
else:
    if self.tilt > -90:
        self.tilt -= self.ROT_VEL
```

Obr. 12 Nastavení sklonu ptáčka v metodě *move* ve třídě Bird

Metoda ***def draw(self, win)*** slouží k vykreslení ptáčka na okno aplikace. Pomocí proměnné *ANIMATION_TIME* si zjistíme, kdy použít jaký obrázek ptáčka, abychom dosáhli jednoduché animace ptáčka. Toho dosáhneme tak, že budeme porovnávat, jestli je hodnota *self.img_count* menší jako hodnota *ANIMATION_TIME* a následně přiřazovat proměnné *self.img* odpovídající obrázky z listu *self.IMGS*. Dále kontrolujeme, jestli ptáček padá, což znamená, že jeho sklon je 90 stupňů, pokud je tohle pravda tak přiřadíme proměnné *self.img* obrázek *self.IMGS[1]*, což je ptáček, který nemává křídly nahoru nebo dolů.

```
self.img_count += 1
# loop through 3 images for bird animation
if self.img_count <= self.ANIMATION_TIME:
    self.img = self.IMGS[0]
```

Obr. 13 Ukázka přiřazení obrázku pro animaci ptáčka v metodě *draw* ve třídě Bird

Poslední, co uděláme v metodě *draw* je vykreslení ptáčka na okno aplikace. Prvně musíme náš obrázek otočit kolem jeho středu, a to uděláme následovně pomocí *pygame.transform.rotate(self.img, self.tilt)*, kde předáváme obrázek, který chceme otočit a sklon (úhel sklonu). Následovně musíme posunout tento vytvořený obrázek do středu okna, a to uděláme pomocí *get_rect()* a vykreslíme pomocí *win.blit.*

```
# tilt the bird
rotated_image = pygame.transform.rotate(self.img, self.tilt)
new_rect = rotated_image.get_rect(center=self.img.get_rect(topleft=(self.x, self.y)).center)
win.blit(rotated_image, new_rect.topleft)
```

Obr. 14 Vykreslení ptáčka v metodě *draw* ve třídě Bird

Poslední metodou ve třídě Bird je ***def get_mask(self)***, kterou budeme později používat při kolizi s trubkami. Jediné, co tato metoda dělá je, že nám vrátí masku z našeho obrázku ptáčka. To znamená, že nastaví všechny průhledné pixely, aby šel ptáček vidět a nenastaví žádné neprůhledné pixely v obrázku. Toho dosáhneme pomocí *pygame.mask.from_surface(self.img).*

5.3.3.2 Třída Pipe

Třída Pipe, reprezentuje trubky a má metody *set_height, move, draw, collide*. Prvně si vytvoříme dvě proměnné a to GAP (mezera mezi trubkami) s hodnotou 200 a VEL (rychlost jakou se trubky pohybují) s hodnotou 5.

Objekt trubky inicializujeme pomocí metody ***def __init__(self, x)***, která má dvě vstupní proměnné a to *self*, která představuje objekt třídy Pipe a *x*, což je horizontální pozice. Definujeme si proměnné *self.x* s hodnotou *x*, *self.height* s hodnotou 0, *self.top* s hodnotou 0, *self.bottom* s hodnotou 0, *self.PIPE_BOTTOM* s hodnotou *PIPE_IMG* (globální proměnná, která má obrázek trubky), *self.PIPE_TOP* pomocí *pygame.transform.flip(PIPE_IMG, False, True),* což nám obrátí obrázek trubky, dále *self.passed* (využijeme pro kolizi a AI, sleduje jestli ptáček proletěl trubkami nebo ne) s hodnotou *False a* na konci inicializace zavoláme funkci *set_height()*, která nám nastaví, kde je horní a spodní trubka a jak jsou veliké.

Metoda ***def set_height(self)***, jak už jsem popsal nám nastaví, kde se budou naše trubky nacházet. Pozici horní trubky si nastavíme náhodně a to tak, že odečteme náhodnou hodnotu, kterou dostaneme pomocí *random.randrange(50, 450)* od výšky horní trubky. Pozici spodní trubky dostaneme tak, že přičteme proměnnou GAP k výšce, kterou jsme dostali náhodně.

```
1 usage
def set_height(self):
    self.height = random.randrange( start: 50,  stop: 450)
    self.top = self.height - self.PIPE_TOP.get_height()
    self.bottom = self.height + self.GAP
```

Obr. 15 Metoda *set_height* ve třídě Pipe

Metoda ***def move(self)*** nám zajistí pohyb trubek. Všechno, co v této metodě uděláme je odečtení *self.VEL* od *self.x*, což nám posune trubky horizontálně doleva.

Metoda ***def draw(self, win)***, která nám slouží k vykreslení trubek na okno aplikace.

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

Obr. 16 Metoda *draw* ve třídě Pipe

Metoda ***def collide(self, bird)*** slouží ke kontrole kolize mezi ptáčkem a trubkami. Vstupní proměnnou je ptáček, u kterého chceme kolizi kontrolovat. Na začátku si získáme masku ptáčka a to pomocí *bird.get_mask()*. Následně musíme získat masku pro horní a spodní trubky a toho dosáhneme pomocí *pygame.mask.from_surface(self.PIPE_TOP)* a *pygame.mask.from_surface(self.PIPE_BOTTOM)*. Následně si musíme vypočítat vzdálenost ptáčka od horní a spodní trubky a jako poslední zjistíme, jestli se pixely ptáčka překrývají s pixely trubky. Pokud ano metoda nám vrátí *True*, pokud ne vrátí *False*.

```
def collide(self, bird):
    bird_mask = bird.get_mask()
    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 t_point or b_point:
        return True

    return False
```

Obr. 17 Metoda *collide* ve třídě Pipe

5.3.3.3 Třída Base

Třída Base, reprezentuje objekt země. Má dvě metody: *move* a *draw*. Na začátku vytvoříme proměnné třídy, a to *VEL* s hodnotou 5, *IMG*, která obsahuje obrázek země *BASE_IMG* a poslední *WIDTH*, což je šířka obrázku *BASE_IMG.get_width()*.

Inicializujeme objekt metodou ***def __init__(self, y)***. Vstupní proměnnou je *y*, která reprezentuje vertikální pozici. Dále definujeme proměnné *self.y* s hodnotou *y*, *self.x1* s hodnotou 0 a *self.x2*, do které přiřadíme šířku našeho obrázku *self.WIDTH.*

Metoda ***def move(self)*** nám slouží k pohybu země. Prvně od *self.x1 a self.x2* odečteme *self.VEL*, tím se budeme pohybovat směrem doleva. Metoda funguje tak, že máme dva obrázky přímo za sebou a posunujeme je na ose x doleva a jakmile se první obrázek dostane mimo okno aplikace, tak se neposunuje dál doleva ale posuneme ho za obrázek druhý. Tím dáváme dojem animace.

```
1 usage
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
```

Obr. 18 Metoda *move* ve třídě Base

Metoda ***def draw(self, win)*** slouží k vykreslení země na okno aplikace.

```
3 usages (3 dynamic)
def draw(self, win):
    win.blit(self.IMG, (self.x1, self.y))
    win.blit(self.IMG, (self.x2, self.y))
```

Obr. 19 Metoda *draw* ve třídě Base

5.3.4 Konfigurační soubor Feedforward pro NEAT algoritmus

První sekci v tomto konfiguračním souboru je sekce NEAT, která specifikuje parametry specifické pro algoritmus NEAT. Tato sekce je vždy povinná. V této sekci nastavíme *fitness_criterion* na hodnotu *max*, to znamená, že z každé generace vezmeme ptáčky, kteří dosáhli nejvyššího skóre, jinak řečeno dosáhli nejvyššího fitness a využijeme je pro mutaci ptáčků do další generace, *fitness_treshold* nastavíme na hodnotu 100, to znamená, pokud nějaký ptáček v generaci dosáhne fitness skóre 100 program se ukončí, *pop_size* nastavíme na 20 a to nám říká kolik ptáčků bude v každé generaci.

Další sekcí je DefaultGenome, která nám slouží pro nastavení parametrů genomu, se kterými každý ptáček v generaci bude začínat. NEAT rozděluje genomy na soubor genů: geny uzlové reprezentující neurony a geny spojovací reprezentující synapse. Nastavíme si naši aktivační funkci, kterou chceme používat *activation_default = tanh,* tanh je hyperbolická funkce, která komprimuje vstupní hodnoty v rozsahu -1 do 1, pomocí této funkce zjistíme v hlavním programu, jestli má ptáček skočit nebo ne. V této sekci musíme také nastavit, kolik budeme mít vstupních, skrytých a výstupních neuronů: *num_hidden* (skryté) s hodnotou 0, *num_inputs* (vstupní) s hodnotou 3 (pozice ptáčka, pozice horní a spodní trubky) a *num_outputs* (výstupní) s hodnotou 1 (jestli má ptáček skočit nebo ne).

Sekce DefaultSpeciesSet, nám definuje parametry související s tím, jak NEAT seskupuje genomy do species (druhů) během evoluce. Nastavíme jediný parametr, který nám říká, jestli ptáčci patří ke stejnému druhu nebo ne, *compatibility_treshold* s hodnotou 3, to znamená, pokud má ptáček menší hodnotu, než je tato prahová hodnota, tak patří ke stejnému druhu.

Sekce DefaultStagnation, definuje parametry, podle kterých kontrolujeme, jestli se zvyšuje fitness v nových generací. Nastavíme *max_stagnation* na hodnotu 20, to znamená, že druhy, které neprojevují zlepšení ve více než tomto počtu generací budou považováný za neprospívající a odstraněny. Dále *species_elitism* nastavíme na hodnotu 2, které nám zajistí, že dva druhy, které neprospívají nebudou odstraněny i když přesáhnou daný počet generací, pokud mají nejvyšší fitness skóre ze všech druhů.

Poslední sekcí je DefaultReproduction, která definuje parametry, které řídí, jak jsou během evoluce vytvářeny nové genomy pro další generaci. Nastavíme tady *elitism* na hodnotu 2, který nám dává počet nejvýkonnějších genomů ze současné generace,

u kterých je zaručeno, že budou bez úprav přeneseny do další generace*, survival_treshold* nastavíme na hodnotu 0,2, tento parametr nám dává podíl jedinců každého druhu, kteří se mnohou rozmnožovat v každé generaci.

5.3.5 Hlavní funkce a pomocné funkce

Pomocná funkce ***def draw_window(win, birds, pipes, base, score, gen)*** nám slouží k vykreslení okna aplikace, kde použijeme všechny doposud vytvořené funkce *draw*. Má šest vstupních proměnných *win*, která reprezentuje šířku a výšku okna aplikace, *birds*, která obsahuje všechny ptáčky, *pipes*, která bude mít všechny trubky, *base*, což je naše země, *score*, do které přičítáme naše skóre neboli jak daleko se ptáček dostal a jako poslední *gen*, což nám bude říkat v jaké generaci našeho NEAT algoritmu se nacházíme.

Na začátku se zeptáme, pokud má *gen* hodnotu 0, tak si pro vizuální účely nastavíme proměnnou *gen* na hodnotu 1. Dále pomocí dvou *for* cyklů budeme procházet jak trubky, tak ptáčky a postupně je vykreslovat.

```
1 usage
def draw_window(win, birds, pipes, base, score, gen):
    if gen == 0:
        gen = 1
    win.blit(BG_IMG, (0, 0))
    for pipe in pipes:
        pipe.draw(win)
    base.draw(win)
    for bird in birds:
        bird.draw(win)
```

Obr. 20 Vykreslení trubek a ptáčků na okno aplikace v pomocné funkci *draw_window*

Poslední věc, co uděláme je, že přidáme text se skórem, počtem generací a počtem, který nám řekne kolik ptáčků je naživu. Toho dosáhneme pomocí využití *render()*, kde jako první argument předáme text, který chceme zobrazit, hodnotu 1 nebo 0 pro antialias, který pokud dáme 1 nám vyhladí hrany textu a jako poslední argument bude barva, který musíme před v RGB kódu, např 255, 255, 255 pro bílou barvu.

```
# score
text = STAT_FONT.render("Score: " + str(score), antialias: 1, color: (255, 255, 255))
win.blit(text, (WIN_WIDTH - text.get_width() - 15, 10))
```

Obr. 21 Ukázka vykreslení skóre na okno aplikace ve funkci *draw_window*

Než se dostaneme k hlavní funkci musíme zajistit načtení našeho konfiguračního souboru pro NEAT algoritmus. To uděláme tak, že zjistíme, kde se konfigurační soubor nachází pomocí podmínky *if __name__ == „__main““* a v této podmínce prvně najdeme cestu ke složce ve které se konfigurační soubor nachází, a to pomocí *local_dir = os.path.dirname(__file__)*, do proměnné *config_path* si uložíme cestu k našemu souboru pomocí *os.path.join(),* kde první argument je složka, kde se nacházíme *local_dir* a druhý argument je název konfiguračního souboru *config-feedforward.txt*. Zavoláme funkci *run()* a předáme ji náš konfigurační soubor *config_path*.

```
if __name__ == "__main__":
    # Determine path to configuration file
    local_dir = os.path.dirname(__file__)
    config_path = os.path.join(local_dir, "config-feedforward.txt")
    run(config_path)
```

Obr. 22 Nalezení cesty ke konfiguračnímu souboru

Funkce ***def run(config)*** nám načte náš konfigurační soubor a bude spouštět funkci *main*, ve které bude samotná hra Flappy Bird. Načtení konfiguračního souboru provedeme následovně pomocí modulu neat *neat.config.Config()*, jako argumenty budeme předávat všechny sekce v konfiguračním souboru plus cestu ke konfiguračnímu souboru.

```
def run(config_file):
    config = neat.config.Config(neat.DefaultGenome, neat.DefaultReproduction, neat.DefaultSpeciesSet,
                                neat.DefaultStagnation, config_file)
```

Obr. 23 Načtení konfiguračního souboru ve funkci *run*

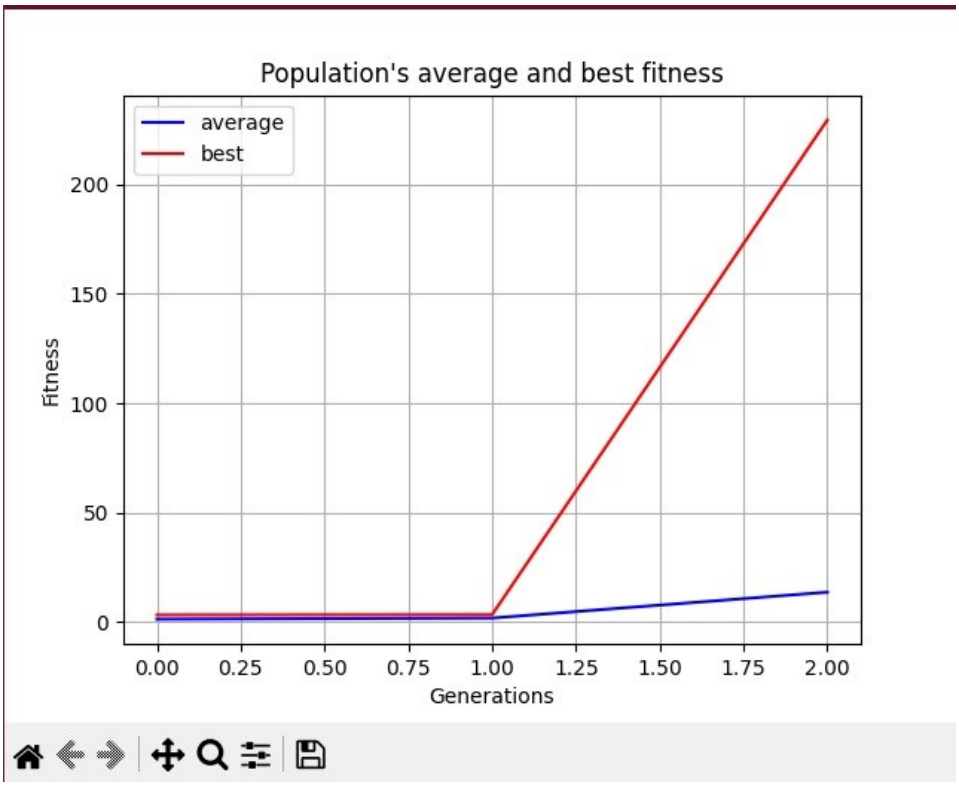
Vytvoříme populaci do proměnné *p* pomocí *neat.Population(config)*. Dále vytvoříme statistiku, která se bude zobrazovat v terminálu a bude nám říkat např. velikost populace, průměrné fitness, nejlepší fitness v dané generaci a další.

```
# Create the population
p = neat.Population(config)
# Add a stdout reporter to show progress in the terminal
p.add_reporter(neat.StdOutReporter(True))
stats = neat.StatisticsReporter()
p.add_reporter(stats)
```

Obr. 24 Vytvoření populace a statistiky ve funkci *run*

Zavoláme naši aktivační funkci pomocí *p.run()*, kde první parametr bude aktivační funkce (v našem případě funkce *main*) a počet generací, kolik chceme, aby tato funkce proběhla.

Nakonec si vykreslíme výsledky do grafu. Na grafu, který se jmenuje Population's average and best fitness (Průměrné a nejlepší fitness populace) se podíváme na nejlepší fitness v dané generaci a na průměrné fitness během průběhu generací. Na ose x máme vypsaný počet generací a na ose y hodnoty fitness. Červená reprezentuje nejlepší fitness a modrá průměrné fitness.



Obr. 25 Graf výsledků

Funkce ***def main (genomes, config)*** je naše hlavní funkce, která obsahuje logiku hry Flappy Bird a přiřazování fitness ptáčkům pro náš NEAT algoritmus. Slouží jako aktivační funkce NEAT algoritmu. Má dvě vstupní proměnné: *genomes*, která jednoduše řečeno je naše aktuální generace ptáčků a *config*, co je náš konfigurační soubor pro NEAT algoritmus.

Vytvoříme si tři prázdné listy: *nets* reprezentující neuronovou síť každého ptáčka, *ge* reprezentující genomy a *birds* reprezentující objekt ptáčka. Listy naplníme pomocí *for* cyklu, kde budeme procházet vstupní *genomes*. Nastavíme defaultní hodnotu fitness každého genomu na hodnotu 0, vytvoříme neuronovou síť pomocí *neat.nn.FeedForwardNetwork()*, kde první parametr bude *genome* a druhý náš konfigurační soubor *config*. Do listu *nets* přidáme na konec vytvořenou neuronovou síť pomocí *nets.append(net)*, do listu *birds* přidáme ptáčka

birds.append(Bird(230, 350)), kde první hodnota je pozice x a druhá pozice y, a do listu *ge* přidáme genome *ge.append(genome)*.

```
nets = []
birds = []
ge = []
for genome_id, genome in genomes:
    genome.fitness = 0
    net = neat.nn.FeedForwardNetwork.create(genome, config)
    nets.append(net)
    birds.append(Bird( x: 230,  y: 350))
    ge.append(genome)
```

Obr. 26 Vytvoření a naplnění listů ve funkci *main*

Hlavní *while* cyklus poběží do té doby, dokud je proměnná *run True* (okno aplikace je otevřené) a dokud máme naživu alespoň jednoho ptáčka. Kontrolu, jestli je okno otevřené provedeme pomocí kontrolu eventu, to znamená, pokud klikneme na křížek okna aplikace.

Než začneme manipulovat ptáčka musíme si zjistit, jakou trubku použít pro aktivační funkci, jelikož je možné, že na displeji je více než jedna. To zjistíme pomocí *if* podmínky, kde se ptáme, jestli velikost listu *pipes* je větší než 1, čímž kontrolujeme, jestli je na displeji více než 1 trubka, a dále se ptáme, jestli pozice ptáčka na ose x je větší než pozice horní trubky na ose x, v podstatě, jestli jsme proletěli trubkou, pokud ano tak do proměnné *pipe_ind* dáme hodnotu 1, čímž v aktivační funkci budeme používat druhou trubku na displeji, pokud ne tak proměnná *pipe_ind* bude mít hodnotu 0.

```
pipe_ind = 0
if len(birds) > 0:
    # determine whether to use first or second pipe for neural network input
    if len(pipes) > 1 and birds[0].x > pipes[0].x + pipes[0].PIPE_TOP.get_width():
        pipe_ind = 1
```

Obr. 27 Podmínka, která určí použití první nebo druhé trubky na displeji pro aktivační funkci

Teď když víme, jakou trubku použít pro aktivační funkci, tak ji můžeme vypočítat. Vytvoříme si *for* cyklus, který bude procházet listem ptáčků *birds*. V tomto *for* cyklu budeme přidávat 0,1 fitness každému ptáčku za každý snímek, co zůstane naživu a budeme pohybovat ptáčka pomocí funkce *bird.move()*. Pro aktivační funkci využijeme tři vstupní argumenty:

pozice ptáčka na ose y *bird.y*, pozici horní trubky na ose y, kterou vypočítáme odečtením pozice ptáčka *bird.y* od výšky horní trubky *pipes[pipe_ind].height* a poslední argument předáme pozici spodní trubky na ose y, kterou vypočítáme odečtením pozice ptáčka *bird.y* od pozice spodní trubky *pipes[pipe_ind].bottom*. Používáme aktivační funkci tanh, takže výsledek bude v rozmezí -1 až 1, pokud bude výsledek větší jako 0,5 tak ptáček skočí.

```
for x, bird in enumerate(birds):  # give each bird a fitness of 0.1 for each frame it stays alive
    bird.move()
    ge[x].fitness += 0.1

    # send bird location, top pipe location, bottom pipe location to determine whether bird jumps or not
    output = nets[birds.index(bird)].activate(
        (bird.y, abs(bird.y - pipes[pipe_ind].height), abs(bird.y - pipes[pipe_ind].bottom)))

    # using tanh as activation function so result will be between -1 and 1, if over 0.5 jump
    if output[0] > 0.5:
        bird.jump()
```

Obr. 28 Výpočet aktivační funkce a rozhodnutí skoku ptáčka

Dále budeme odstraňovat fitness, pokud ptáček narazil do trubky. Toho dosáhneme pomocí *for* cyklu a *if* podmínky. Budeme procházet list ptáčků *birds* a kontrolovat, jestli ptáček narazil do trubky pomocí funkce *pipe.collide(bird)*, pokud ano odečteme od genomu daného ptáčka 5 fitness a odstraníme ho z listů *nets, ge, birds.* Také si zjistíme, jestli máme přidat další trubku, pokud ptáček proletěl mezi trubkami a to tak, že se ptáme, jestli je není pravda, že ptáček proletěl mezi trubkami a že pozice trubky na ose x je menší než pozice ptáčka na ose x.

```
rem = []
for pipe in pipes:
    pipe.move()
    # check for collision
    for bird in birds:
        if pipe.collide(bird):
            ge[birds.index(bird)].fitness -= 5
            nets.pop(birds.index(bird))
            ge.pop(birds.index(bird))
            birds.pop(birds.index(bird))

    # if pipe is not on screen append to a list to remove
    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
```

Obr. 29 Kontrola kolize a kontrola, jestli ptáček proletěl mezi trubkami

Následně budeme přidávat trubky a zvyšovat odměnu fitness a obtížnost a to tak, že budeme zmenšovat mezeru mezi následujícími trubkami. Pokud jsme proletěli mezi trubkami, znamená to, že musíme zvýšit skóre o 1 a přidal další trubky mezi, kterými ptáček musí proletět. Pomocí jednoduché *if* podmínky se ptáme, jestli ptáček proletěl mezi trubkami, pokud ano, tak ke skóre přičteme 1 a dále se ptáme, jestli mezera mezi trubkami je větší jak 600 px, jestli je větší, tak přidáme fitness k genomu, na konec listu *pipes* přidáme další trubku a zmenšíme mezeru o 20 px. Pokud bude mezera mezi trubkami větší jak 600 px vždycky přičteme genomu 1 fitness, když je mezera menší jak 600 px, přičteme 1,5 fitness a když je mezera rovna 500 px přestaneme zmenšovat mezeru mezi trubkami a přičteme 3 fitness.

```
if add_pipe:
    score += 1
    if gap_pipes > 600:
        # give bird 1 fitness for passing through a pipe
        for genome in ge:
            genome.fitness += 1
        pipes.append(Pipe(gap_pipes))
        gap_pipes -= 20
    elif gap_pipes <= 600 and gap_pipes != 500:
        for genome in ge:
            genome.fitness += 1.5
        pipes.append(Pipe(gap_pipes))
        gap_pipes -= 20
    elif gap_pipes == 500:
        for genome in ge:
            genome.fitness += 3
        pipes.append(Pipe(gap_pipes))
```

Obr. 30 Přidávání trubek a zvýšení obtížnosti

Také musíme kontrolovat, jestli ptáček spadl na zem anebo jestli skákal nahoru, a tím se dostal mimo okno aplikace. Toho dosáhneme jednoduché *if* podmínky, kde se ptáme, jestli pozice ptáčka je větší, než velikost země anebo pokud pozice ptáčka je menší jako -50 px, což znamená, že ptáček vyletěl příliš vysoko, tím pádem je mimo okno aplikace. Pokud je jedna z těchto podmínek pravdivá, tak se odečte od ptáčka 2 fitness a odstraníme ptáčka z listů *nets, birds, ge*.

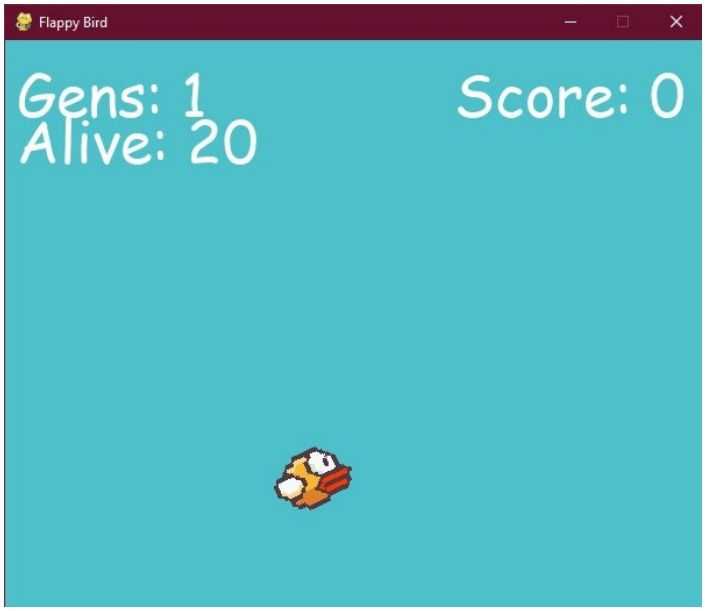
```
for bird in birds:
    if bird.y + bird.img.get_height() - 10 >= FLOOR or bird.y < -50:
        ge[birds.index(bird)].fitness -= 2
        nets.pop(birds.index(bird))
        ge.pop(birds.index(bird))
        birds.pop(birds.index(bird))
```

Obr. 31 Kontrola, jestli ptáček zemřel vyletěním mimo okno nebo spadnutím na zem

5.4 Praktické testování implementace

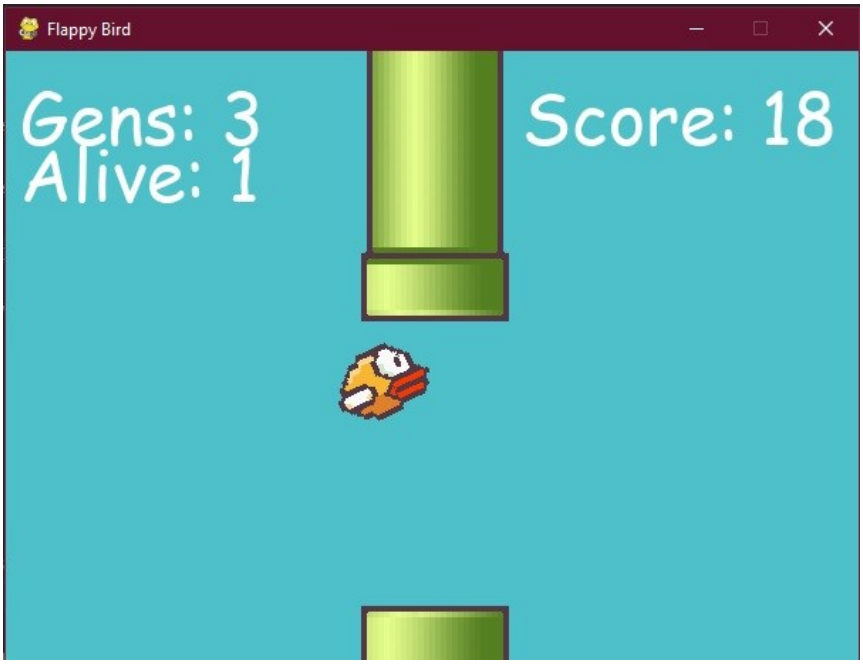
Na začátku nám NEAT algoritmus vytvoří generaci o populaci 20 ptáčků, kteří budou hrát hru Flappy Bird. Všichni ptáčci dostávají fitness, pokud neumřou, a to za prolétnutí mezi trubky a zůstáním naživu. Ptáčci s nejvyšším skórem (fitness) neboli ti, kteří zůstali naživu nejdéle se stanou rodiči nové generace. Nová generace zdědí vlastnosti od ptáčků z generace předchozí a vylepšuje je. Toho se dosáhne pomocí mutace neuronové sítě ptáčků. Tímto způsobem dosáhneme toho, že za určitý počet generací dostaneme ptáčka/ptáčky, kteří teoreticky můžou prolétávat trubkami do nekonečna.

Po spuštění se nám otevře okno s názvem Flappy Bird a můžeme vidět, skóre, jakého jsme dosáhli v aktuální generaci, počet živých ptáčků a počet celkových generací. V první generaci naše AI, ještě neví, aby dostalo, co největší množství fitness musí zůstat naživu co nejdéle a prolétávat mezi trubkami.



Obr. 32 Začátek hry Flappy Bird

Hned jak začneme hrát, většina ptáčků vyletí příliš vysoko anebo spadne na zem. Jakmile, ale alespoň jeden ptáček proletí mezi trubkami, tak další generace už vědí, že aby dosáhli většího fitness, tak musí prolétnout mezi trubkami, a proto jich zůstane naživu více a déle. Jak můžeme vidět na obrázku (Obr. 33) s populací 20 ptáčků vznikne mutace, která dokáže získat vysoké skóre relativně rychle.



Obr. 33 Průběh hry Flappy Bird, vytrénovaný ptáček

V terminálu se nám vypisuje statistika, každé generace. Co tady můžeme zjistit je například, průměrné fitness populace, nejlepší fitness v dané generaci, čas, jak dlouho generace trvala a další. Mean genetic distance (průměrná genetická vzdálenost) je důležitá míra pro pochopení rozmanitosti populace neuronových sítí. Čím větší tato hodnota bude, tím lepší a úspěšnější je algoritmus.

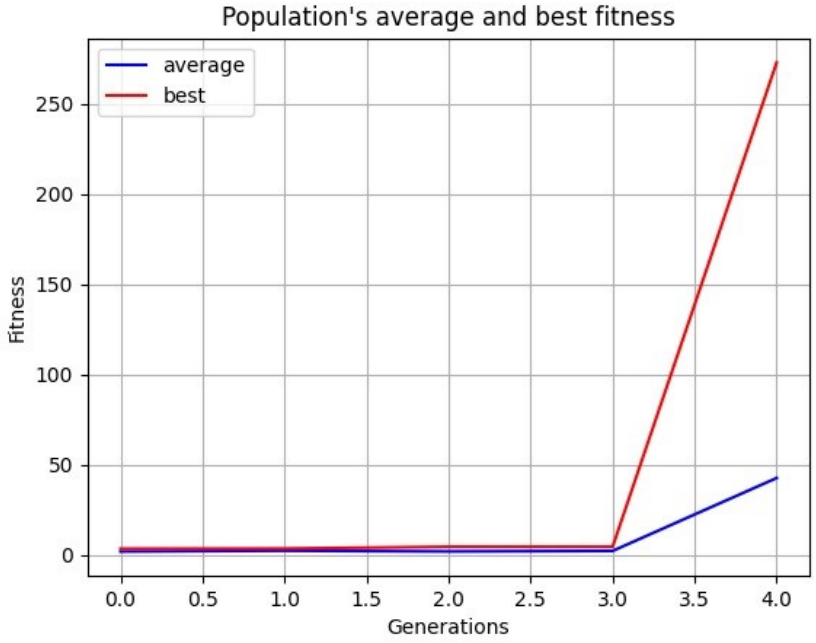
```
Population's average fitness: 2.85500 stdev: 2.80704
Best fitness: 14.00000 - size: (1, 3) - species 1 - id 4
Average adjusted fitness: 0.181
Mean genetic distance 1.480, standard deviation 0.609
Population of 20 members in 1 species:
   ID   age  size  fitness  adj fit  stag
  ====  ===  ====  =======  =======  ====
     1    1    20     14.0    0.181     0
Total extinctions: 0
Generation time: 1.392 sec (1.192 average)
```

Obr. 34 Statistika v terminálu

5.5 Vyhodnocení výsledků

Porovnáme si hru Flappy Bird, kdy mezi trubkami zůstává pořád stejná vzdálenost a další verzi, kdy se mezera mezi trubkami pomalu zmenšuje. Ptáčci poletí do té doby, dokud nedosáhnou fitness alespoň 100 nebo skóre alespoň 25.

Prvně se podíváme na hru, kdy se mezera mezi trubkami nezmenšuje. Na grafu vidíme, že ptáčci dosáhli vysoké hodnoty fitness relativně rychle, už ve čtvrté generaci ptáček získal více jak 250 fitness.



Obr. 35 Graf normální hry Flappy Bird

Na obrázku (viz. Obr. 36) můžeme vidět nejlepší genome v generaci 4, který dosáhl nejvyššího fitness. Složitost (1, 2) (*Complexity*) nám značí kolik uzlů (nodes) a spojení (connections) má genome, v tomto případě genome obsahuje jeden uzel a dvě spojení. Genome má klíč 78 a dosáhl fitness 272,999. Bias (práh) v uzlu nám posouvá celou aktivační funkci doleva (pro negativní bias) nebo doprava (pro pozitivní bias). To umožňuje neuronu aktivovat se (dosáhnout určité výstupní hodnoty) i když je vážený součet vstupů nulový. Aktivační funkce uzlu je tanh, což je hyperbolický tangens, který po výpočtu vrátí hodnotu v rozmezí -1 až 1. Dále vidíme, že máme dvě spojení. První spojení má klíč (-3, 0) a váhu kladnou 0,7095, druhé spojení má klíč (-2, 0) a váhu zápornou -0,6964. To znamená, že signál z uzlu -2 bude mít tendenci aktivovat uzel 0, zatímco signál z uzlu -3 bude mít tendenci uzel 0 tlumit.

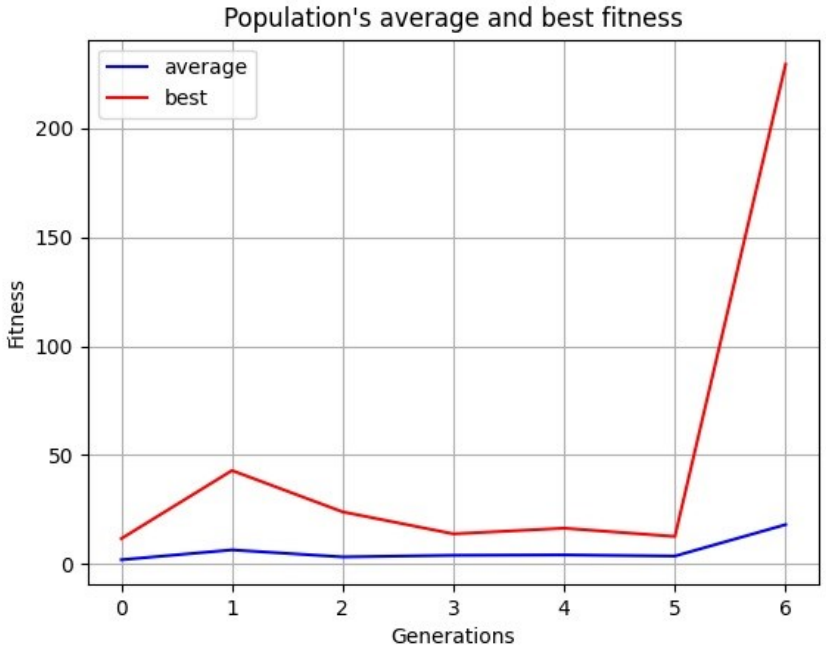
```
Best individual in generation 4 meets fitness threshold - complexity: (1, 2)

Best genome:
Key: 78
Fitness: 272.9999999999941
Nodes:
    0 DefaultNodeGene(key=0, bias=-0.3159538784813808, response=1.0, activation=tanh, aggregation=sum)
Connections:
    DefaultConnectionGene(key=(-3, 0), weight=-0.696443983084864, enabled=True)
    DefaultConnectionGene(key=(-2, 0), weight=0.7095414191724786, enabled=True)
```

Obr. 36 Statistika nejlepšího genomu v normální hře Flappy Bird

Následně se podíváme na graf a statistiku hry, kde se obtížnost zvýší a to tím, že budeme zmenšovat mezery mezi trubkami.

Uvidíme na grafu (viz. Obr. 37), že při těžší obtížnosti ptáčkům zabralo skoro dvakrát tolik generací, aby dosáhli přibližně stejného fitness jako u normální hry. Na rozdíl od normální hry, kdy fitness zůstalo na stejné hodnotě ve všech generací kromě poslední, můžeme vidět, že při zvýšené obtížnosti se fitness měnilo během generací.



Obr. 37 Graf hry Flappy Bird se zvýšenou obtížností

Ve statistice (viz. Obr. 38) můžeme vidět, že nejlepší genome byl v šesté generaci se složitostí (*Complexity*) (1, 2). Genome má klíč (id) 119 a dosáhl fitness 229,449. Můžeme vidět, že počet uzlů a spojení se nijak nezměnil od normální hry. Jedinou změnou je, že máme tři spojení, ale z toho jedno, které vede z uzlu -1 do uzlu 0 je neaktivní, to znamená, že nemá

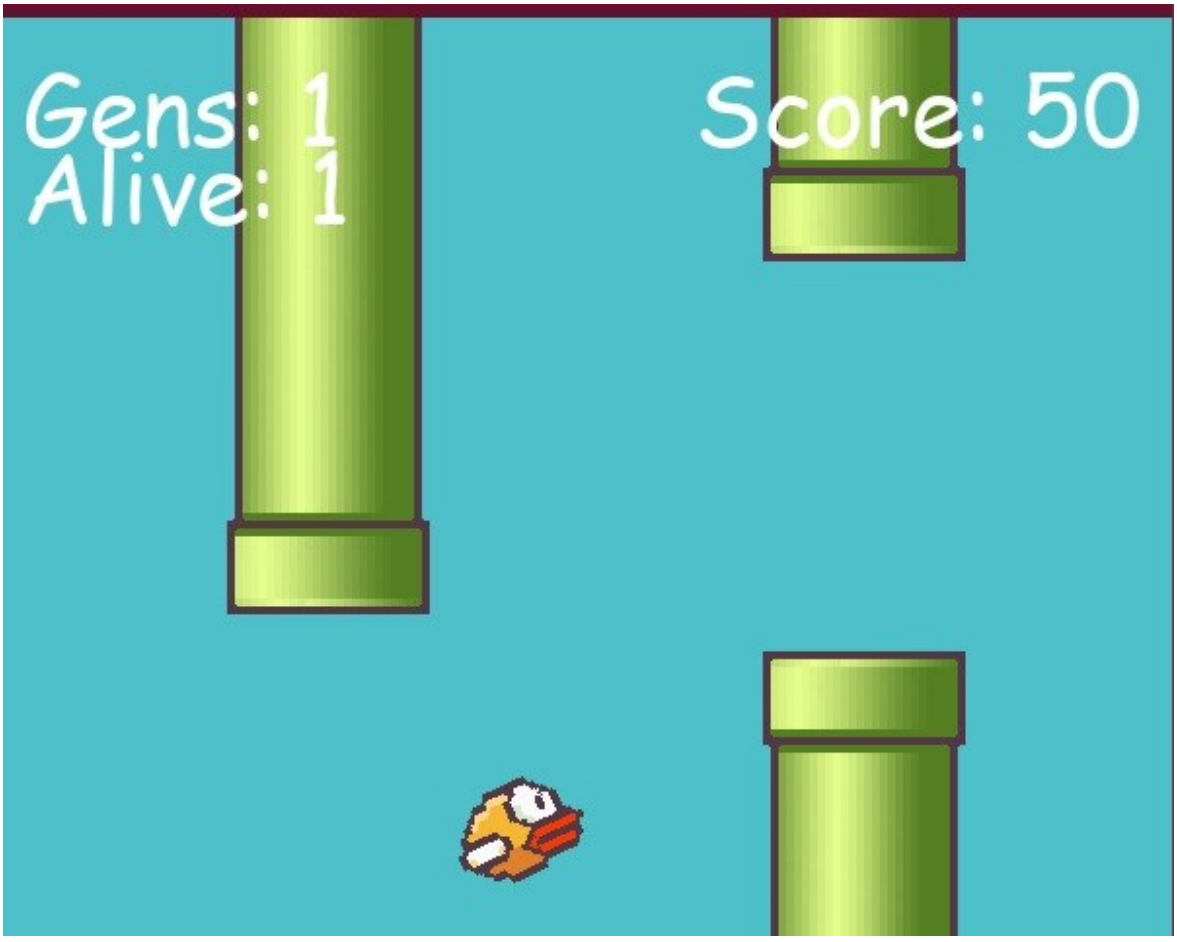
vliv na uzel s klíčem 0. Neaktivní spojení rozeznáme od aktivních pomocí poslední vlastnosti *enabled*, která bude mít hodnotu *False* pro neaktivní a *True* pro aktivní.

```
Best individual in generation 6 meets fitness threshold - complexity: (1, 2)

Best genome:
Key: 119
Fitness: 229.49999999999423
Nodes:
    0 DefaultNodeGene(key=0, bias=-0.3660754897760333, response=1.0, activation=tanh, aggregation=sum)
Connections:
    DefaultConnectionGene(key=(-3, 0), weight=-0.9485980108759232, enabled=True)
    DefaultConnectionGene(key=(-2, 0), weight=0.5494513525416923, enabled=True)
    DefaultConnectionGene(key=(-1, 0), weight=-1.416546891015954, enabled=False)
```

Obr. 38 Statistika nejlepšího genomu v obtížnější hře Flappy Bird

Implementace byla pro účely demonstrace hry naučeného ptáčka lehce poupravena. Načteme si našeho nejlepšího ptáčka a podíváme, jak daleko se dostane. Ptáček byl trénovaný v obtížnější hře.



Obr. 39 Hra Flappy Bird s naučeným ptáčkem

Vidíme, že naučený ptáček, si vedl o mnoho lépe, dosáhl skóre 50, a fitness dokonce až 730 (viz. Obr. 40).

```
Best genome:
Key: 44
Fitness: 730.1000000000639
Nodes:
    0 DefaultNodeGene(key=0, bias=-0.8529682290404772, response=1.0, activation=tanh, aggregation=sum)
Connections:
    DefaultConnectionGene(key=(-3, 0), weight=-1.2744441145111767, enabled=True)
    DefaultConnectionGene(key=(-2, 0), weight=1.3844411298581156, enabled=True)
```

Obr. 40 Statistika naučeného ptáčka obtížnější hry Flappy Bird

6 SNAKE

Tato kapitola se zabývá mobilní hrou Snake. Kapitola začíná popisem samotné hry a principem jejího fungování. Dále se zaměřuje na implementaci algoritmu Q-learning a na vytvoření simulovaného prostředí hry Snake pomocí Pythonu.

6.1 Popis a herní principy

Snake, známá také jako Had, je klasická videohra, která se proslavila v 80. letech 20. století na herních konzolích a mobilních telefonech. Hra je dodnes populární a existuje mnoho variant s různými herními mechanismy a bonusy. [32]

Základní myšlenka:

- Hráč ovládá hada, který se pohybuje po herním poli.
- Cílem hry je nasbírat co nejvíce bodů snědením jídla, které se náhodně objevuje na herním poli.
- Po snědení jídla se ocas hada prodlouží o jeden segment.
- Hráč nesmí narazit do stěn herního pole ani do vlastního těla hada.
- Pokud k tomu dojde, hráč končí.

Herní principy:

- Hráč ovládá hada pomocí šipek na displeji, klávesnici nebo tlačítek na ovladači.
- Had se pohybuje po herním poli v jednom směru, dokud hráč nezadá změnu směru.
- Jídlo se objevuje na náhodných místech na hřišti a po snědení zvětší ocas hada.
- Na herním poli se mohou nacházet i bonusy, které můžou mít různé efekty, například zrychlení hada, prodloužení ocasu o více segmentů, nebo teleportaci jídla na jiné místo.
- Obtížnost hry se obvykle zvyšuje s postupujícím časem, čímž se zrychluje had a zkracuje se reakční doba hráče.

6.2 PyTorch

PyTorch je knihovna pro strojové učení založená na knihovně Torch. PyTorch se používá pro různé aplikace, jako je počítačové vidění a zpracování přirozeného jazyka. Původně ji vyvinula společnost Meta AI (dříve Facebook AI) a nyní je součástí nadace Linux Foundation. [39]

Klíčové vlastnosti PyTorch:

- **Výpočty s tenzory:** PyTorch nabízí podobné funkce jako NumPy pro práci s tenzory, což jsou vícerozměrná pole dat, která jsou základním stavebním prvkem pro výpočty v deep learningu. PyTorch navíc umí využívat výkon grafických karet (GPU) pro výrazné zrychlení výpočtů. [39]
- **Dynamické neuronové sítě:** PyTorch umožňuje vytvářet a trénovat neuronové sítě pomocí automatické diferenciace. To znamená, že PyTorch může automaticky vypočítávat gradienty funkcí, což je nezbytné pro optimalizaci modelů během fáze učení. [39]
- **Flexibilita a snadné použití:** PyTorch je známý svou flexibilitou a relativní snadností použití. Umožňuje psát kód v Pythonu a nabízí nástroje pro přechod mezi různými režimy spouštění (eager a graph mode), což poskytuje výhody obou. [39]

6.3 Implementace

Implementace herního prostředí Snake a metody Q-learning proběhla pomocí YouTube tutoriálu. [38]

Implementace je rozdělená do tří python skriptů:

- **snake_game:** skript obsahující logiku hry Snake a herní prostředí, obsahuje funkce *reset, _place_food, play_step, is_collision, _update_ui a _move*.
- **agent:** skript, který obsahuje agenta, který se učí hrát hru Snake, obsahuje funkce *get_state, remember, train_long_memory, train_short_memory, get_action, train*.
- **model:** obsahuje základní komponenty pro výcvik agenta pomocí metody deep Q-learning, obsahuje funkce *forward, save, train_step*.

Před samotnou implementací si musím stáhnout dvě knihovny: Pygame a PyTorch. Vytvoříme si nový projekt a do konzole vložíme následující příklad pro instalaci Pygame: pip install pygame. Příkaz na instalaci PyTorch dostaneme následovně, otevřeme si stránku https://pytorch.org/ a zde přejdeme na položku „Get Started“, kde si přizpůsobíme nastavení pro PyTorch, jaké chceme. Budeme chtít nejnovější verzi PyTorch, operační systém si zvolíme podle toho, co používáme, balíček si zvolíme pip a jazyk python, poslední si zvolíme CPU jako naši výpočetní platformu. To nám zobrazí příkaz, který zadáme do terminálu: pip install torch torchvision.

Obr. 41 Příklad nastavení pro instalaci PyTorch

6.4 Skript snake_game

Používané moduly:

- *pygame:* pro vytvoření prostředí hry Snake.
- *random:* pro generování náhodných čísel.
- *enum:* modul, ze kterého importujeme třídu Enum, která se využívá pro definování sady konstant.
- *collections:* modul, ze kterého importujeme funkci namedtuple, která vytvoří podtřídu klasické n-tice s pojmenovanými prvky.
- *numpy:* modul používaný pro výpočty.

6.4.1 Globální proměnné

Vytvoříme si n-tici pomocí *namedtuple(),* kde první argument bude jméno naší n-tice, v tomto případě *Point* a druhý argument budou prvky *x a y*, které budou reprezentovat osu x a osu y. Definujeme si barvy pomocí kódu RGB *WHITE = (255, 255, 255)*, které budeme později využívat na vykreslení hada, ovoce a textu. *BLOCK_SIZE*, která nám definuje velikost těla hada. Poslední proměnná *SPEED*, která nám říká kolik snímků za sekundu hra poběží.

```
Point = namedtuple( typename: 'Point', field_names: 'x, y')

# rgb colors
WHITE = (255, 255, 255)
RED = (200, 0, 0)
BLUE1 = (0, 0, 255)
BLUE2 = (0, 100, 255)
BLACK = (0, 0, 0)
BLOCK_SIZE = 20
SPEED = 4000
```

Obr. 42 Globální proměnné hry Snake

6.4.2 Vytvoření tříd a jejich metod

6.4.2.1 Třída Direction

Vytvoříme se třídu Enum, která se jmenuje Direction. V této třídě si definujeme pohybu našeho hada: *RIGHT, LEFT, UP, DOWN.*

```
class Direction(Enum):
    RIGHT = 1
    LEFT = 2
    UP = 3
    DOWN = 4
```

Obr. 43 Třída Direction hry Snake

6.4.2.2 Třída SnakeGameAI

Inicializujeme třídu pomocí ***def __init__(self, w=640, h=480)***, kde *w* bude šířka a *h* výška okna aplikace. Definujeme si výšku a šířku pomocí *self.h a self.w*, dále displej *pygame.display.set_mode()*, kde argumenty budou *self.w* a *self.h* a *self.clock* pomocí *pygame.time.Clock()*. Nakonec zavoláme metodu *self.reset()*, která nám inicializuje herní stav.

Metoda ***def reset(self)*** se stará o inicializaci herního stavu. Nastavíme si směr, kterým se had bude na začátku pohybovat *self.direction = Direction.RIGHT*, pomocí naší n-tice si vytvoříme tělo hada. Hlavu hada si definujeme jako *Point(self.w/ 2, self.h/ 2)*, tak aby výsledný had byl uprostřed okna aplikace při startu. Celé tělo hada bude mít tři bloky, které vytvoříme

pomocí listu, kde na nultém indexu bude vytvořená hlava hada *self.head*, na prvním indexu bude blok na stejné pozici *self.y* ale od pozice *self.x* odečteme *BLOCK_SIZE*, na druhém indexu bude poslední blok těla hada na stejné pozici *self.y* a od pozice *self.x* se odečte dvojnásobek *BLOCK_SIZE*. *Self.score* nastavíme na hodnotu 0, *self.food* na hodnotu *NONE.* Následně zavoláme funkci *self._place_food()*, která nám vygeneruje ovoce a proměnnou *self.frame_iteration* nastavíme na hodnotu 0.

```
def reset(self):
    # init game state
    self.direction = Direction.RIGHT

    self.head = Point(self.w / 2, self.h / 2)
    self.snake = [self.head,
                  Point(self.head.x - BLOCK_SIZE, self.head.y),
                  Point(self.head.x - (2 * BLOCK_SIZE), self.head.y)]

    self.score = 0
    self.food = None
    self._place_food()
    self.frame_iteration = 0
```

Obr. 44 Inicializace herního stavu hry Snake

Metoda ***def _place_food(self)*****,** nám bude generovat ovoce pokaždé, když ho had sní. Náhodně si vygenerujeme pozici na ose x a y pomocí *random.randint()*. Následně pomocí naší n-tice ho přiřadíme k proměnné *self.food*. Nakonec si pomocí podmínky *if*, zjistíme, jestli had snědl ovoce, pokud ano, tak znovu zavoláme metodu *self._place_food()*.

```
def _place_food(self):
    x = random.randint( a: 0, (self.w - BLOCK_SIZE) // BLOCK_SIZE) * BLOCK_SIZE
    y = random.randint( a: 0, (self.h - BLOCK_SIZE) // BLOCK_SIZE) * BLOCK_SIZE
    self.food = Point(x, y)
    if self.food in self.snake:
        self._place_food()
```

Obr. 45 Metoda *_place_food* hry Snake

Metoda ***def play_step(self, action)*****,** je hlavní metoda, ve které budeme kontrolovat, jestli hra skončila, had měl kolizi se stěnou nebo samým sebou, budeme pohybovat hadem a dávat mu odměnu za úspěšné snězení ovoce nebo trest (odečtení odměny) za kolizi.

Prvně nastavíme odměnu *reward* na hodnotu 0 a přičteme 1 k *self.frame_iteration*. Než začneme s pohybem hada zkontrolujeme, jestli je okno aplikace otevřené pomocí eventu.

```
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        pygame.quit()
        quit()
```

Obr. 46 Kontrola, jestli je otevřené okno aplikace hry Snake

Pomocí *self._move(action)*, pohneme hada ve směru *action* a musíme aktualizovat pozici hlavy v těle hada, to uděláme pomocí *self.snake.insert(0, self.head)*, kde na index 0 vložíme hlavu hada. Použijeme *if* podmínku, kde se ptáme, jestli hra trvala déle, než je velikost hada vynásobená stem, pokud ano, tak hra skončila nastavíme *game_over* na hodnotu *True*, od odměny *reward* odečteme 10 a vrátíme *return reward, game_over, self.score.* Dále zkontrolujeme kolizi pomocí *if self.is_collision()*, pokud ano, tak hra skončila. Nastavíme *game_over* na hodnotu *True*, od odměny *reward* odečteme 20 a vrátíme *return reward, game_over, self.score*.

```
game_over = False
if self.frame_iteration > 100*len(self.snake):
    game_over = True
    reward -= 10
    return reward, game_over, self.score

if self.is_collision():
    game_over = True
    reward -= 20
    return reward, game_over, self.score
```

Obr. 47 Kontrola, jestli hra Snake skončila

Dále se ptáme, jestli had snědl ovoce, pokud ano, tak zvedneme skóre o 1 a k odměně přičteme 10 a vygenerujeme nové ovoce pomocí *self._place_food()*. Pokud ne, tak odstraníme z listu *self.snake* na posledním indexu blok těla, to znamená, že se had jenom pohnul a nesnědl žádné ovoce. Nakonec pomocí metody *self._update_ui()* aktualizujeme okno

aplikace (hada, ovoce, skóre) a také aktualizujeme rychlost hry pomocí *self.clock.tick(SPEED)*, kde předáváme parametr *SPEED* a vracíme *return reward, game_over, self.score.*

```
# 3. place new food or just move
if self.head == self.food:
    self.score += 1
    reward += 10
    self._place_food()
else:
    self.snake.pop()

# 4. update ui and clock
self._update_ui()
self.clock.tick(SPEED)
# 5 return reward, game over and score
return reward, game_over, self.score
```

Obr. 48 Kontrola snězení ovoce, aktualizace okna aplikace a rychlost hry Snake

Metoda ***def is_collision(self, pt=None)*** kontroluje, jestli had narazil do stěn nebo do sebe samotného. Na začátku se ptáme, jestli proměnná *pt* je *None*, pokud ano, tak ji definujeme jako *pt = self.head*. Kolizi se stěnou kontrolujeme tak, že se ptáme, jestli had se nachází mimo okno aplikace na pozici *x* nebo *y*, pokud ano, tak vracíme *True*. Dále se ptáme, jestli had narazil sám do sebe a to tak, že zjistíme, jestli hlava hada se nachází v těle hada, pokud ano, vracíme *True*. Pokud ani jedna z podmínek není pravda, tak nám metoda vrací *False.*

```
def is_collision(self, pt=None):
    if pt is None:
        pt = self.head
    # hits boundary
    if pt.x > self.w - BLOCK_SIZE or pt.x < 0 or pt.y > self.h - BLOCK_SIZE or pt.y < 0:
        return True
    # hits itself
    if pt in self.snake[1:]:
        return True

    return False
```

Obr. 49 Kontrola kolize hada hry Snake

Metoda ***def _update_ui(self)*** nám slouží k vytvoření okna aplikace a vykreslení hada, ovoce a textu skóre. Pozadí vyplníme černou barvou, a to pomocí *self.display.fill(BLACK)*. Dále si vykreslíme našeho hada a ovoce pomocí *pygame.draw.rect()*, kde předáváme argumenty *self.display*, kde chceme hada a ovoce vykreslit, barvu hada a ovoce a pomocí *pygame.Rect* předáme, kde se had a ovoce nachází a jejich velikost. Pomocí *font.render()* si vykreslíme skóre do okna aplikace. Nakonec aktualizujeme okno aplikace a všechno, co se uvnitř děje, a to pomocí *pygame.display.flip()*.

```
def _update_ui(self):
    self.display.fill(BLACK)
    for pt in self.snake:
        pygame.draw.rect(self.display, BLUE1, pygame.Rect(pt.x, pt.y, BLOCK_SIZE, BLOCK_SIZE))
        pygame.draw.rect(self.display, BLUE2, pygame.Rect(pt.x + 4, pt.y + 4, 12, 12))

    pygame.draw.rect(self.display, RED, pygame.Rect(self.food.x, self.food.y, BLOCK_SIZE, BLOCK_SIZE))

    text = font.render("Score: " + str(self.score), antialias: True, WHITE)
    self.display.blit(text, dest: [0, 0])
    pygame.display.flip()
```

Obr. 50 Aktualizace okna aplikace hry Snake

Poslední metodou je pohyb hada ***def _move(self, action)***, kde dostaneme list *action*, který se skládá z nul a jediné jedničky, která nám říká, jakým směrem se had bude pohybovat. Pokud se jednička nachází na indexu 0, tak se had pohybuje rovně, pokud na indexu 1, tak se pohybuje doprava a pokud na indexu 2, tak se pohybuje doleva. Vytvoříme si list *clock_wise*, do kterého si uložíme směry pohybu ve směru hodinových ručiček: vpravo, dolů, vlevo, nahoru. Zjistíme si index inicializovaného směru v listu *clock_wise*, pomocí *clock_wise.index()*. Musíme zkontrolovat, jestli list *action*, se rovná, kterým se chceme pohybovat, pokud se pohybujeme rovně, tak se nic nemění a do proměnné *new_direction* přiřadíme směr na indexu, který jsme si zjistili. Pokud se chceme pohybovat směrem doprava musíme si vypočítat nový index a to tak, že přičteme 1 ke stávajícímu indexu *idx* a následným modulováním 4. Tento index použijeme k přiřazení směru z listu *clock_wise[new_idx]* do proměnné *new_direction*. Přejdeme ze směrů doprava do směru dolů (had udělá pravou zatáčku). Pokud se chceme pohybovat směrem doleva vypočítáme si nový index odečtením 1 od stávajícího indexu *idx* a následně modulováním 4. Tento index použijeme k přiřazení směru z listu *clock_wise[new_idx]* do proměnné *new_direction*. Následně přiřadíme nový směr *new_direction* do proměnné *self.direction*.

```
clock_wise = [Direction.RIGHT, Direction.DOWN, Direction.LEFT, Direction.UP]
idx = clock_wise.index(self.direction)

if np.array_equal(action, a2: [1, 0, 0]):
    new_direction = clock_wise[idx]  # no change
elif np.array_equal(action, a2: [0, 1, 0]):
    next_idx = (idx + 1) % 4
    new_direction = clock_wise[next_idx]  # right turn, r -> d -> l -> u
elif np.array_equal(action, a2: [0, 0, 1]):
    next_idx = (idx - 1) % 4
    new_direction = clock_wise[next_idx]  # left turn, r -> u -> l -> d

self.direction = new_direction
```

Obr. 51 Zjištění nového směru pro pohyb hada hry Snake

Nakonec si zjistíme, jakému směru se nový směr rovná pomocí *if* podmínky, např. *self.direction == Direction.RIGHT* a zjistíme, kde přidat na pozici *x* nebo *y* blok hada a aktualizujeme hlavu hada na novou pozici následovně *self.head = Point(x, y)*.

```
x = self.head.x
y = self.head.y
if self.direction == Direction.RIGHT:
    x += BLOCK_SIZE
elif self.direction == Direction.LEFT:
    x -= BLOCK_SIZE
elif self.direction == Direction.DOWN:
    y += BLOCK_SIZE
elif self.direction == Direction.UP:
    y -= BLOCK_SIZE

self.head = Point(x, y)
```

Obr. 52 Pohyb hada novým směrem ve hře Snake

6.5 Skript agent

Používané moduly:

- *torch:* knihovna pro výpočty v deep learningu.
- *random:* pro generování náhodných čísel.
- *numpy:* modul používaný pro výpočty.
- *collections:* modul, ze kterého importujeme deque, datová struktura, která nabízí efektivní přidávání a odebírání prvků z obou konců fronty.
- *snake_game:* skript, ze kterého importujeme třídy SnakeGameAI, Direction, Point.
- *model:* skript, ze kterého importujeme třídy Linear_QNet a Qtrainer.
- *helper:* pomocný skript, ze kterého importujeme funkci plot, která slouží k vytvoření grafu

6.5.1 Globální proměnné

Vytvoříme proměnnou *MAX_MEMORY* s hodnotou 100 000, která definuje maximální počet zkušeností, které si agent uloží do své paměti, obsahuje: stav, akce, odměna, následující stav a dokončeno (říká nám, jestli hra skončila). Proměnná *BATCH_SIZE* s hodnotou 1000 definuje počet zkušeností, které jsou v každé iteraci vzorkování z paměti pro výcvik agenta. Proměnná *LR* (Learning Rate) s hodnotou 0,001, která řídí velikost kroků používaných k aktualizaci vah modelu během procesu učení.

6.5.2 Třída Agent a její metody

Inicializujeme třídu pomocí ***def __init__(self).*** Vytvoříme proměnnou *self.n_games* s hodnotou 0, která sleduje počet odehraných her, *self.epsilon* s hodnotou 0, která řídí náhodnost agenta, *self.gamma* s hodnotou 0,9, používá se k výpočtu budoucí odměny (diskontní faktor), *self.memory* pomocí *deque(maxlen=MAX_MEMORY)*, kde budeme ukládat zkušenosti agenta, pokud přesáhneme délku *MAX_MEMORY*, tak bude nejstarší prvek z levé strany odstraněn pomocí metody *popleft()*. Do proměnné *self.model* si vytvoříme instanci třídy *Linear_QNet()*, která bude mít tři parametry: vstupní, skrytou a výstupní vrstvu neuronové sítě. Vstupní vrstva bude mít hodnotu 11, počet stavů, které budeme používat, skrytá vrstva hodnotu 256 a výstupní vrstva hodnotu 3 (směr, kterým se budeme pohybovat). Proměnná *self.trainer* je instancí třídy Qtrainer, které předáváme tři parametry: *self.model, LR a self.gamma*. Slouží pro trénování neuronové sítě agenta.

```
def __init__(self):
    self.n_games = 0
    self.epsilon = 0  # randomness
    self.gamma = 0.9  # discount rate
    self.memory = deque(maxlen=MAX_MEMORY)  # if exceed -> popleft()
    self.model = Linear_QNet( input_size: 11,  hidden_size: 256,  output_size: 3)
    self.trainer = QTrainer(self.model, lr=LR, gamma=self.gamma)
```

Obr. 53 Inicializace třídy Agent hry Snake

V metodě ***def get_state(self, game)*** si nastavíme všechny stavy hry. První, co musíme udělat je dostat hlavu hada, a to uděláme pomocí *game.snake[0]* a uložíme do proměnné *head*. Následně si vytvoříme n-tice, které nám budou reprezentovat pozice okolo hlavy hada (vpravo, vlevo, dole, nahoře), např. do proměnné *point_l* si uložíme pozici vlevo od hlavy hada pomocí *Point(),* kde první argument bude pozice hlavy hada na ose *head.x* a od ní odečteme 20 a druhý argument bude pozice hlavy hada na ose *head.y.* Dále si zjistíme směry hry, a to pomocí *game.direction*.

```
def get_state(self, game):
    head = game.snake[0]
    point_l = Point(head.x - 20, head.y)
    point_r = Point(head.x + 20, head.y)
    point_u = Point(head.x, head.y - 20)
    point_d = Point(head.x, head.y + 20)

    dir_l = game.direction == Direction.LEFT
    dir_r = game.direction == Direction.RIGHT
    dir_u = game.direction == Direction.UP
    dir_d = game.direction == Direction.DOWN
```

Obr. 54 Vytvoření proměnných pro pozice okolo hlavy hada a směrů hry

Vytvoříme si pole *state = []*, ve které si definujeme všechny stavy hry: nebezpečí vlevo, vpravo, rovně, směry pohybu a pozice ovoce. A to uděláme tak, že pokud aktuální směr pohybu je vpravo a zároveň zavoláme metodu na kontrolu kolize *game.is_collision(point_r)*, kde kontrolujeme kolizi vpravo od hlavy hada, tak to znamená, že je nebezpečí rovně. Obdobně zjistíme, jestli je nebezpečí vpravo a vlevo.

```
state = [
    # Danger straight
    (dir_r and game.is_collision(point_r)) or
    (dir_l and game.is_collision(point_l)) or
    (dir_u and game.is_collision(point_u)) or
    (dir_d and game.is_collision(point_d)),

    # Danger right
    (dir_u and game.is_collision(point_r)) or
    (dir_d and game.is_collision(point_l)) or
    (dir_l and game.is_collision(point_u)) or
    (dir_r and game.is_collision(point_d)),

    # Danger left
    (dir_d and game.is_collision(point_r)) or
    (dir_u and game.is_collision(point_l)) or
    (dir_r and game.is_collision(point_u)) or
    (dir_l and game.is_collision(point_d)),
```

Obr. 55 Stavy pro kontrolu nebezpečí okolo hada

Dále přidáme stavy směru pohybu a stavy, kde se nachází ovoce. Metoda nám vrátí pole.

```
        # Move direction
        dir_l,
        dir_r,
        dir_u,
        dir_d,

        # Food location
        game.food.x < game.head.x,  # food left
        game.food.x > game.head.x,  # food right
        game.food.y < game.head.y,  # food up
        game.food.y > game.head.y  # food down
    ]

    return np.array(state, dtype=int)
```

Obr. 56 Stavy směru pohybu a kde se nachází ovoce

Metoda ***def remember(self, state, action, reward, next_state, done)*** slouží k uložení informací stavu, akce, odměny, nový stav (poté, co agent provedl akci) a jestli hra skončila.

```
def remember(self, state, action, reward, next_state, done):
    self.memory.append((state, action, reward, next_state, done))  # popleft if MAX_MEMORY is reached
```

Obr. 57 Metoda *remember* pro uložení informací hry Snake

Metoda ***def train_long_memory(self)*****,** slouží k trénování Q-sítě agenta pomocí opakování zkušeností z větší dávky minulých zkušeností. Pomocí podmínky *if* kontrolujeme, jestli je velikost *self.memory* větší než velikost *BATCH_SIZE*, pokud ano, tak se náhodně vzorkuje *BATCH_SIZE* zkušeností z deque pomocí *random.sample(self.memory, BATCH_SIZE)*. Tím vytvoříme seznam n-tic, kde každá n-tice představuje zkušenost. Pokud velikost *self.memory* není větší než velikost *BATCH_SIZE*, tak použije metoda pro opakování zkušeností *self.memory*.

Následně rozbalíme tento seznam n-tic do samostatných seznamů pro stavy, akce, odměny, nové stavy a příznaky konce hry pomocí *zip(*mini_sample)*, nakonec s těmito parametry zavoláme metodu *self.trainer.train_step()* k trénování Q-sítě.

```
def train_long_memory(self):
    if len(self.memory) > BATCH_SIZE:
        mini_sample = random.sample(self.memory, BATCH_SIZE)  # list of tuples
    else:
        mini_sample = self.memory

    states, actions, rewards, next_states, dones = zip(*mini_sample)
    self.trainer.train_step(states, actions, rewards, next_states, dones)
```

Obr. 58 Trénování Q-sítě pomocí minulých zkušeností

Metoda ***def train_short_memory(self, state, action, reward, next_state, done)*** se na rozdíl od metody *train_long_memory* zaměřuje na trénování Q-sítě pomocí jediné, nejnovější zkušenosti.

```
def train_short_memory(self, state, action, reward, next_state, done):
    self.trainer.train_step(state, action, reward, next_state, done)
```

Obr. 59 Trénování Q-sítě pomocí nejnovější zkušenosti

Metoda ***def get_action(self, state)*** využívá epsilon-greedy přístup pro nalezení rovnováhy mezi průzkumem (zkoušením nových věcí) a využíváním (volbou akce s nejvyšší předpokládanou odměnou).

Proměnná epsilon je inicializována na 0, ale roste s rostoucím počtem her *self.n_games*, hodnota epsilon představuje pravděpodobnost průzkumu. Generujeme náhodné číslo mezi 0 a 200, pokud je vygenerované číslo menší než epsilon, tak zvolí agent náhodný pohyb hada (vlevo, vpravo, nahoru nebo dolů). Pokud je náhodné číslo větší nebo rovno epsilon, upřednostňuje agent využívání. Převedeme aktuální stav na tensor vhodný pro Q-síť pomocí *torch.tensor(state, dtype=torch.float)*, tento tensor použijeme jako vstup pro model Q-sítě. Q-síť předpoví budoucí odměnu pro každou možnou akci (vpravo, vlevo, nahoru, dolů) a funkce *torch.argmax()* se použije k nalezení indexu akce s nejvyšší předpovězenou odměnou, tato akce je pak zvolená jako finální pohyb. Nakonec metoda vrátí *final_move*, který indikuje zvolenou akci (jeden prvek nastaven na 1 a ostatní na 0).

```
1 usage
def get_action(self, state):
    # random moves: tradeoff exploration  / exploitation
    self.epsilon = 80 - self.n_games
    final_move = [0, 0, 0]
    if random.randint( a: 0,  b: 200) < self.epsilon:
        move = random.randint( a: 0,  b: 2)
        final_move[move] = 1
    else:
        state0 = torch.tensor(state, dtype=torch.float)
        prediction = self.model(state0)
        move = torch.argmax(prediction).item()
        final_move[move] = 1

    return final_move
```

Obr. 60 Metoda *get_action* pro získání finální pohybu pomocí průzkumu a využíváním

Metoda ***def train()*** obsahuje proces tréninku agenta. Vytvoříme si dva listy *plot_scores, plot_mean_scores* s hodnotou 0, které slouží pro ukládání skóre a průměrného skóre v průběhu hry. *Total_score* nastaveno na hodnotu 0 pro akumulaci celkového skóre napříč hrami, *record* má hodnotu 0, pro zobrazení nejvyššího dosaženého skóre. *Agent()* inicializuje třídu Agent a *SnakeGameAI()* inicializuje třídu herního prostředí hada.

Hlavní funkcionalita je v cyklu *while*, který představuje nepřetržité učení, dokud není přerušeno. Použijeme metodu *agent.get_state(game)* pro získání aktuálního stavu hry, metodou *agent.get_action(state_old)* rozhodneme o dalším pohybu, který agent provede v aktuálním stavu. Vybraný pohyb se pošle do herního prostředí pomocí *game.play_step(final_move)* a tato metoda nám vrátí tři hodnoty: *reward*, *done, score*. Následně zavoláme metodu *agent.train_short_memory(state_old, final_move, reward, state_new, done) a agent.remember(state_old, final_move, reward, state_new, done)* pro uložení aktuální zkušenosti pro opakované přehrávání a trénování Q-sítě.

```
while True:
    # get old state
    state_old = agent.get_state(game)

    # get move
    final_move = agent.get_action(state_old)

    # perform move -> get new state
    reward, done, score = game.play_step(final_move)
    state_new = agent.get_state(game)

    # train short memory
    agent.train_short_memory(state_old, final_move, reward, state_new, done)

    # remember
    agent.remember(state_old, final_move, reward, state_new, done)
```

Obr. 61 Cyklus *while* v metodě *train*

Pokud hra skončila *done = True*, zavolá se metoda *game.reset()*, aby se prostředí hry resetovalo pro další kolo. Zavolá se metoda *agent.train_long_memory()*, aby se provedl hlouběji zaměřený tréninkový krok pomocí agentova bufferu pro opakované přehrávání. Pokud je skóre dosažené v aktuální hře větší než rekord, tak nahradíme rekord aktuálním skórem. Pomocí funkce *plot()* vytvoříme graf jednotlivého skóre hry a průměrného skóre po každé hře.

```
if done:
    # train long memory (replay memory), plot result
    game.reset()
    agent.n_games += 1
    agent.train_long_memory()

    if score > record:
        record = score
        agent.model.save()

    print('Game', agent.n_games, 'Score', score, 'Record', record)

    plot_scores.append(score)
    total_score += score
    mean_scores = total_score / agent.n_games
    plot_mean_scores.append(mean_scores)
    plot(plot_scores, plot_mean_scores)
```

Obr. 62 Hra skončila v metodě *train*

6.6 Skript model

Používané moduly:

- *torch:* knihovna pro výpočty v deep learningu.
- *torch.nn:* submodul knihovny PyTorch, který nabízí kolekci předdefinovaných modulů neuronových sítí.
- *torch.optim:* submodul knihovny PyTorch, který poskytuje různé optimalizační algoritmy používané k trénování neuronových sítí.
- *torch.nn.functional:* submodul knihovny PyTorch, který obsahuje sadu běžně používaných aktivačních funkcí a ztrátových funkcí implementovaných jako funkcionální operace.
- *os:* modul, který poskytuje funkce pro interakci s operačním systémem

6.6.1 Vytvoření tříd a jejich metod

6.6.1.1 Třída Linear_QNet

Definujeme třídu Linear_QNet, která dědí od třídy nn.Module *class Linear_QNet(nn.Module).*

Inicializujeme třídu pomocí ***def __init__(self, input_size, hidden_size, output_size)***, kde máme tři argumenty: *input_size* (velikost vstupních dat neuronové sítě), *hidden_size* (velikost skryté vrstvy neuronové sítě), *output_size* (velikost výstupu neuronové sítě). Využijeme *super().__init__()* k zavolání konstruktoru nadřazené třídy (nn.Module) uvnitř odvozené třídy (Linear_QNet). Vytvoříme si dvě lineární vrstvy pomocí *nn.Linear(input_size, hidden_size),* kde transformujeme velikost vstupu na výstup o velikosti *hidden_size*. Druhá vrstva bere výstup první vrstvy *hidden_size* a transformuje jej na výstup o velikosti *output_size.*

```
class Linear_QNet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.linear1 = nn.Linear(input_size, hidden_size)
        self.linear2 = nn.Linear(hidden_size, output_size)
```

Obr. 63 Inicializace třídy *Linear_QNet*

Metoda ***def forwar(self, x)*** definuje, jak data procházejí sítí během průchodu vpřed. Bere vstup *x a* vrací konečný výstup sítě. Vstup *x* prochází první lineární vrstvou, výsledek se poté prochází aktivační funkcí ReLU *F.relu(self.linear(x))*. Výstup z aktivační funkce prochází druhou lineární vrstvou *self.linear2(x)*.

```
def forward(self, x):
    x = F.relu(self.linear1(x))
    x = self.linear2(x)
    return x
```

Obr. 64 Metoda *forward*

Metoda ***def save(self, file_name='model.pth')*** umožňuje uložit parametry trénovaného modelu do souboru. Volitelný argument *file_name* nám určí název souboru. Definujeme si cestu ke složce, kam se model uloží *model_folder_path*. Zkontrolujeme, jestli cesta ke složce existuje, pokud ne, tak ji vytvoříme *os.makedirs(model_folder_path)*.

Vytvoříme úplnou cestu k souboru, kam se model uloží sloučením cesty ke složce a názvu souboru pomocí *os.path.join()*. Nakonec uložíme pomocí *torch.save()*.

```
def save(self, file_name='model.pth'):
    model_folder_path = './model'
    if not os.path.exists(model_folder_path):
        os.makedirs(model_folder_path)

    file_name = os.path.join(model_folder_path, file_name)
    torch.save(self.state_dict(), file_name)
```

Obr. 65 Metoda *save* pro uložení parametrů trénovaného modulu

6.6.1.2 Třída Qtrainer

Inicializujeme třídu ***def __init__(self, model, lr, gamma)***, která přebírá tři argumenty. *Model* je neuronová síť, která se bude trénovat, *lr* je míra učení (Learning Rate) pro optimalizátor, *gamma* je diskontní faktor. Vytvoříme optimalizátor Adam pro parametry modelu s danou mírou učení *optim.Adam()* a definujeme funkci ztráty jako střední kvadratickou chybu (Mean Squared Error) *nn.MSELoss()*.

```
class QTrainer:
    def __init__(self, model, lr, gamma):
        self.lr = lr
        self.gamma = gamma
        self.model = model
        self.optimizer = optim.Adam(model.parameters(), lr=self.lr)
        self.criterion = nn.MSELoss()
```

Obr. 66 Inicializace třídy *QTtrainer*

Metoda ***def train_step(self, state, action, reward, next_state, done)*** definuje tréninkovou smyčku pro agenta Q-learningu. Vstupní argumenty *state, action, reward, next_state, done* převedeme na tensory pomocí *torch.tensor()*, které představují numerická data používaná pro výpočty uvnitř modelu. Kontrolujeme, jestli tensor stavu má jednu dimenzi, což představuje jediný vzorek, modely DQN však obvykle pro efektivitu pracují s dávkami zkušeností. Pokud má tedy tensor stavu jednu dimenzi, funkce *torch.unsqueeze(state, 0)* vloží novou dimenzi na zadaný index, v tomto případě na index 0. Tím převedeme tensor s jedním vzorkem na dávku s jediným prvek, to také uděláme pro *next_state, action, reward*. Hodnotu

proměnné *done* zabalíme do n-tice pomocí *done = (done,)*, čím zajistíme stejnou strukturu jako ostatní tensory v dávce.

```
def train_step(self, state, action, reward, next_state, done):
    state = torch.tensor(state, dtype=torch.float)
    next_state = torch.tensor(next_state, dtype=torch.float)
    action = torch.tensor(action, dtype=torch.long)
    reward = torch.tensor(reward, dtype=torch.float)
    # (n, x)

    if len(state.shape) == 1:
        # (1, x)
        state = torch.unsqueeze(state, dim: 0)
        next_state = torch.unsqueeze(next_state, dim: 0)
        action = torch.unsqueeze(action, dim: 0)
        reward = torch.unsqueeze(reward, dim: 0)
        done = (done,)
```

Obr. 67 Převod argumentů metody *train_step* na tensory a na dávky s jediným prvkem

Předpovídáme Q-hodnoty pro aktuální stav pomocí modelu agenta *self.model(state)* a vytvoříme kopii předpovězených hodnot *pred.clone()*, která bude použita jako cíl tréninku modelu.

Využijeme *for* cyklu pro výpočet cílových Q-hodnot, bude procházet každou zkušeností v dávce. Uvnitř cyklu inicializujeme *Q_new* s okamžitou odměnou získanou za aktuální zkušenost. Kontrolujeme, zda zkušenost skončila pomocí podmínky *if not done[idx]*, pokud zkušenost neskončila aplikujeme Bellmanovu rovnici k výpočtu cílové Q-hodnoty. Následně aktualizujeme cílovou Q-hodnotu pro konkrétní akci provedenou v dané zkušenosti *target[idx][torch.argmax(action[idx]).item()] = Q_new.*

Vyčistíme všechny existující gradienty z předchozího kroku tréninku před výpočtem ztráty *self.optimizer.zero_grad().* Vypočítáme chybu mezi predikcemi modelu a požadovanými Q-hodnotami *self.criterion(target, pred)*. Následně provedeme zpětné šíření, což nám vypočítá gradienty pro každý parametr, které ukazují, kolik každý parametr přispěl k chybě. Nakonec optimalizátor provede krok k aktualizaci parametrů modelu na základě vypočtených gradientů.

```
pred = self.model(state)
target = pred.clone()
for idx in range(len(done)):
    Q_new = reward[idx]
    if not done[idx]:
        Q_new = reward[idx] + self.gamma * torch.max(self.model(next_state[idx]))

    target[idx][torch.argmax(action[idx]).item()] = Q_new
self.optimizer.zero_grad()
loss = self.criterion(target, pred)
loss.backward()
self.optimizer.step()
```

Obr. 68 Predikce Q-hodnot, výpočet cílových Q-hodnot, výpočet ztráty a optimalizace

6.7 Praktické testování implementace

Snake se učí hrát hru v Deep Q-learningu tím, že zkoumá herní prostředí a dostává odměny za své akce.

V každém kroku hry je stav hry reprezentován souborem informací, které popisují aktuální situaci: pozici jídla, směr, kterým se had pohybuje a nebezpečí blízko hada. Agent má k dispozici omezený počet akcí, které může provést v každém kroku: pohnout se nahoru, dolů, doleva, doprava.

Pro každou kombinaci stavu hry a akce si agent udržuje hodnotu Q, která odráží očekávanou budoucí odměnu za provedení této akce v daném stavu. Q-hodnoty tvoří neuronovou síť, ze které se agent učí. V každém kroku hry si agent vybere akci k provedení. V programu využíváme epsilon-greedy algoritmu, to znamená, že na začátku agent volí akce náhodně, aby prozkoumal různé možnosti. Postupně ale začne dávat přednost akcím s vyšší Q-hodnotou, které slibují větší odměnu.

Po provedení akce, obdržení odměny a pozorování nového stavu hry se Q-hodnoty aktualizují pomocí metody zvané Bellmanova rovnice, která zohledňuje okamžitou odměnu a očekávanou budoucí odměnu, a pomáhá agentovi naučit se dlouhodobě prospěšné strategie.

Proces výběru akce, získávání odměny a aktualizace Q-hodnot se opakuje mnoho her. Postupně se tak Q-hodnoty zpřesňují a agent se učí vybírat ty akce, které vedou k lepším výsledkům (delší had, vyšší skóre).

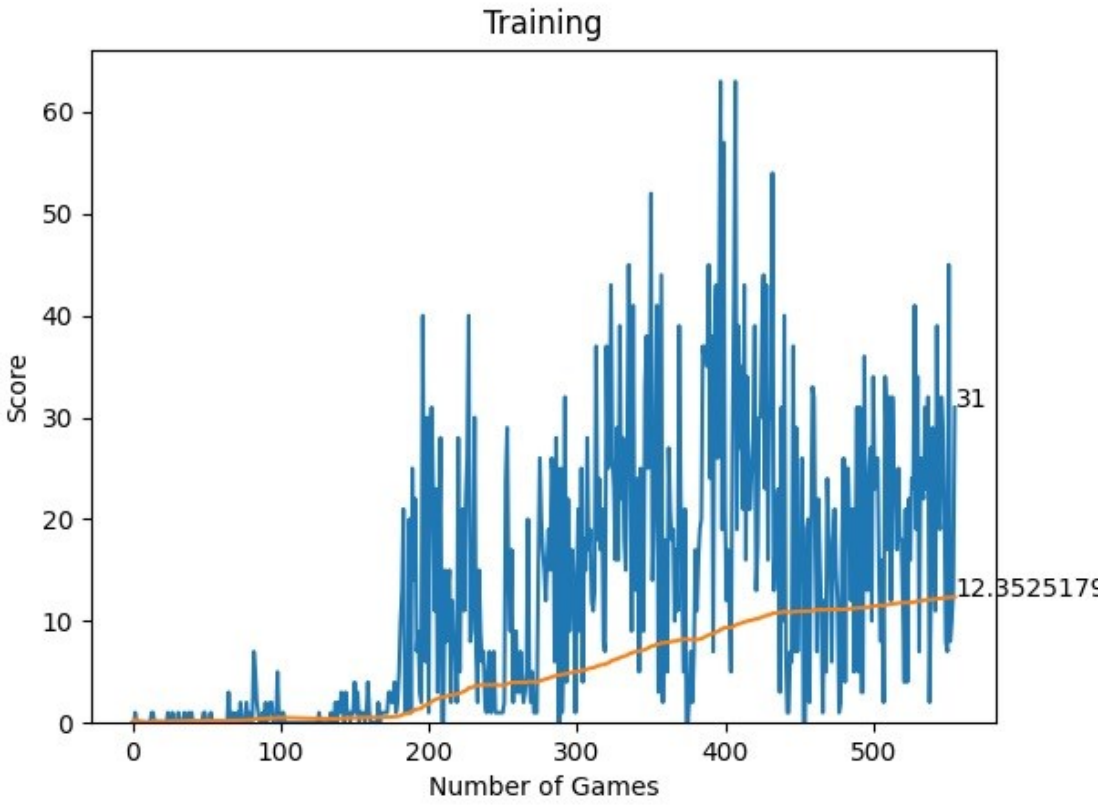
6.8 Vyhodnocení výsledků

Na grafu (viz. Obr. 67) můžeme vidět počet odehraných her (osa x) a skóre (osa y). Počet dosaženého skóre v každé hře je vyznačeno v grafu modrou barvou a průměrný počet skóre v průběhu her je vyznačen oranžovou barvou.

Vidíme, že proces tréninku v tomto případě zabral okolo 550 her. Prvních 180 her můžeme vidět, že had prozkoumává herní prostředí a učí se. Následně využije znalosti herního prostředí, aby dosáhl, co největší skóre. Můžeme vidět, že postupem času had dosáhl skóre většího než 60 okolo hry 400.

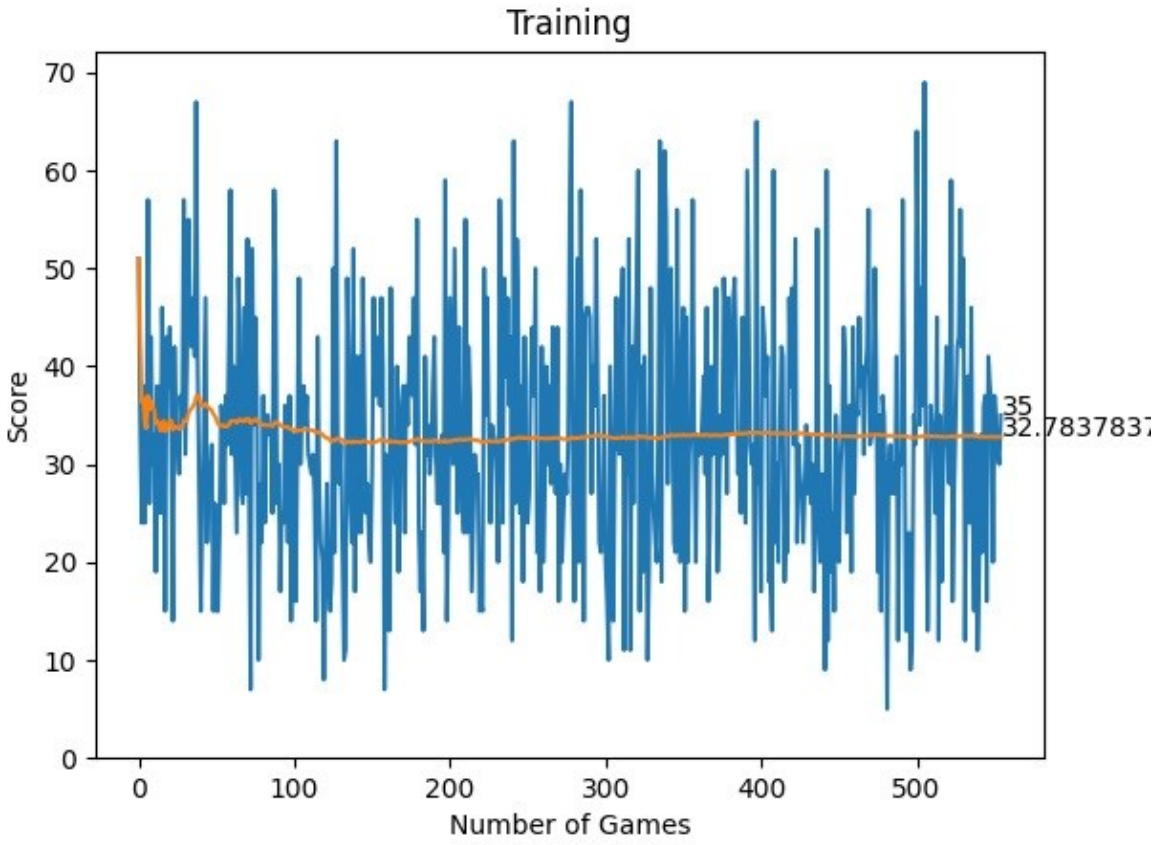
Můžeme vidět, že po dosažení nejvyššího skóre se had pohyboval v rozmezí skóre 20 až 40. To nastává z různých důvodů, agent může upřednostňovat využívání osvědčených strategií před průzkumem nových. Nedostatkem tréninkových dat, jednoduše řečeno, pro agenta nemusí být 550 her dostatek dat k prozkoumání všech strategií.

Tohle by se dalo vyřešit laděním hyperparametrů jako jsou: míra učení, míra průzkumu (epsilon) nebo struktura odměn, abychom podpořili více průzkum nebo upřednostnili akce vedoucí k vyšším skóre.



Obr. 69 Graf výsledků hry Snake

Podíváme se na hru naučeného modelu hada. Implementace byla pro účely demonstrace lehce poupravena. Na grafu (viz. Obr. 70) vidíme, že jsme odehrály okolo 550 her. Průměrné skóre se udržuje na 32.78 snězených ovocí za hru. Naučený had dosáhl největšího skóre 70 a můžeme také vidět, že většinu času dosahoval více než průměrného skóre.



Obr. 70 Graf výsledků naučeného modelu hada

ZÁVĚR

Umělá inteligence se stala nedílnou součástí herního průmyslu, a to i v kontextu populárních mobilních her. Tato práce zkoumala různé metody umělé inteligence a jejich aplikaci v mobilních hrách, a to jak teoreticky, tak i prakticky. Analýza historie a metod umělé inteligence poskytla hlubší porozumění základům tohoto oboru a jeho vývoji v herním průmyslu.

Rozbor metod umělé inteligence provedený v teoretické části této práce ukázal bohatou škálu způsobů, které lze využít pro vytváření inteligetních herních systémů. Diskuse o možnostech a limitech implementace umělé inteligence v mobilních hrách zdůraznila důležitost vyváženého přístupu k vývoji her s ohledem na technická omezení, ale také na etické a sociální aspekty. Analýza etických aspektů využití umělé inteligence v herním prostředí zdůraznila potřebu zohlednit dopady her na hráče a společnost jako celek.

Praktická část této práce demonstrovala konkrétní implementace algoritmů NEAT a Q-learningu v mobilních hrách Flappy Bird a Snake. Tyto implementace ukázaly schopnost umělé inteligence přizpůsobit se různému hernímu prostředí a optimalizovat strategie na základě interakce s uživatelem.

Celkově lze konstatovat, že umělá inteligence má obrovský potenciál přinést inovace a zlepšení do světa mobilních her, ale současně jsou předmětem diskuse důležité otázky týkající se jejího využití a dopadů. Budoucí vývoj v tomto směru bude vyžadovat neustálou diskusi, regulaci a uvědomění si odpovědnosti tvůrců her vůči uživatelům a společnosti jako celku.

Je nezbytné, aby tvůrci her pokračovali v prozkoumávání etických standardů a praktik, které by měly řídit vývoj a implementaci umělé inteligence v herním prostředí. Spolupráce mezi vývojáři, akademickou sférou, regulátory a veřejností bude klíčová pro dosažení vyváženého a udržitelného vývoje v této oblasti.

SEZNAM POUŽITÉ LITERATURY

[1] LUKÁŠ, Ondřej, ULRICH, Herbert a Vojtěch JINDRA, ed. Stručná historie umělé inteligence. *AI dětem* [online]. 2024 [cit. 2024-05-05]. Dostupné z: https://aidetem.cz/obecny-uvod-do-umele-inteligence/strucna-historie-umele-inteligence/

[2] Anon. Historie umělé inteligence. *Machine Learning College* [online]. (nedatováno) [cit. 2024-05-05]. Dostupné z: https://www.mlcollege.com/historie-umele-inteligence/

[3] PIERCE, David. From Eliza to ChatGPT: why people spent 60 years building chatbots. *The Verge* [online]. 2024 [cit. 2024-05-05]. Dostupné z: https://www.theverge.com/24054603/chatbot-chatgpt-eliza-history-ai-assistants-video

[4] Anon. Mycin. *Autoblocks* [online]. (nedatováno) [cit. 2024-05-05]. Dostupné z: https://www.autoblocks.ai/glossary/mycin

[5] IBM. Deep Blue. *IBM* [online]. (nedatováno) [cit. 2024-05-05]. Dostupné z: https://www.ibm.com/history/deep-blue

[6] ABI FADEL, Chadi. The New PageRank of AI: Unraveling the Revolution of the Transformer Model in Language Processing. *LinkedIn* [online]. 2023 [cit. 2024-05-05]. Dostupné z: https://www.linkedin.com/pulse/new-pagerank-ai-unraveling-revolution-transformer-model-abi-fadel

[7] D. FOOTE, Keith. A Brief History of Machine Learning. *DATAVERSITY* [online]. 2021 [cit. 2024-05-05]. Dostupné z: https://www.dataversity.net/a-brief-history-of-machine-learning/

[8] SAP. Co je to strojové učení? *SAP* [online]. 2023 [cit. 2024-05-05]. Dostupné z: https://www.sap.com/cz/products/artificial-intelligence/what-is-machine-learning.html

[9] ROSER, Max, Hannah RITCHIE a Edouard MATHIEU. What is Moore's Law? *Our World in Data* [online]. 2023 [cit. 2024-05-05]. Dostupné z: https://ourworldindata.org/moores-law

[10] AIContentfy. Exploring the Future: The Rise of Google Search Voice and Its Implications. *AIContentfy* [online]. 2023 [cit. 2024-05-05]. Dostupné z: https://aicontentfy.com/en/blog/exploring-future-rise-of-google-search-voice-and-its-implications

[11] Coursera. What Is OpenAI? Everything You Need To Know. *Coursera* [online]. 2024 [cit. 2024-05-05]. Dostupné z: https://www.coursera.org/articles/what-is-openai

[12] Anon. Finite State Machine: Mealy State Machine and Moore State Machine. *ElProCus* [online]. (nedatováno) [cit. 2024-05-05]. Dostupné z: https://www.elprocus.com/finite-state-machine-mealy-state-machine-and-moore-state-machine/

[13] MIĆOVIĆ, Nemanja. Paper Insight: Learning of Behavior Trees for Autonomous Agents. *Nordeus Engineering* [online]. 2020 [cit. 2024-05-05]. Dostupné z: https://engineering.nordeus.com/learning-of-behavior-trees-for-autonomous-agents/

[14] SIMPSON, Chris. Behavior trees for AI: How they work. *Game Developer* [online]. 2024 [cit. 2024-05-05]. Dostupné z: https://www.gamedeveloper.com/programming/behavior-trees-for-ai-how-they-work

[15] PARVEEZ, Saniya a Roberto IRIONDO. Decision Trees in Machine Learning (ML) with Python Tutorial. *Towards AI* [online]. 2021 [cit. 2024-05-05]. Dostupné z: https://pub.towardsai.net/decision-trees-in-machine-learning-ml-with-python-tutorial-3bfb457bce67

[16] GRAHAM, Ross, Hugh MCCABE a Stephen SHERIDAN. Pathfinding in Computer Games. *The ITB Journal*. 2003, 1-26.

[17] IBM. What is machine learning (ML)? *IBM* [online]. (nedatováno) [cit. 2024-05-05]. Dostupné z: https://www.ibm.com/topics/machine-learning

[18] IBM. What is supervised learning? *IBM* [online]. (nedatováno) [cit. 2024-05-05]. Dostupné z: https://www.ibm.com/topics/supervised-learning

[19] STECANELLA, Bruno. *Support Vector Machines (SVM) Algorithm Explained* [online]. 2017 [cit. 2024-05-05]. Dostupné z: https://monkeylearn.com/blog/introduction-to-support-vector-machines-svm/

[20] IBM. What is random forest? *IBM* [online]. (nedatováno) [cit. 2024-05-05]. Dostupné z: https://www.ibm.com/topics/random-forest

[21] IBM. What is logistic regression. *IBM* [online]. (nedatováno) [cit. 2024-05-05]. Dostupné z: https://www.ibm.com/topics/logistic-regression?mhsrc=ibmsearch_a&mhq=logistic%20regression

[22] IBM. What is unsupervised learning? *IBM* [online]. (nedatováno) [cit. 2024-05-05]. Dostupné z: https://www.ibm.com/topics/unsupervised-learning

[23] BHATT, Shweta. Reinforcement Learning 101. *Towards Data Science* [online]. 2018 [cit. 2024-05-05]. Dostupné z: https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292

[24] IBM. What is a neural network? *IBM* [online]. (nedatováno) [cit. 2024-05-05]. Dostupné z: https://www.ibm.com/topics/neural-networks

[25] STANLEY, K.O. a MIIKKULAINEN, R. Efficient evolution of neural network topologies. Online. In: *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)*. IEEE, 2002, s. 1757-1762. ISBN 0-7803-7282-4. Dostupné z: https://doi.org/10.1109/CEC.2002.1004508. [cit. 2024-05-05].

[26] GROSAN, Crina. *Intelligent systems: a modern approach*. Intelligent systems reference library. Berlin: Springer, 2011. ISBN 978-3-642-21004-4. [cit. 2024-05-05].

[27] Anon. Behind The Scenes: How Game Developers Use AI in Mobile Games. *Juego Studio* [online]. 2024 [cit. 2024-05-05]. Dostupné z: https://www.juegostudio.com/blog/how-game-developers-use-ai-in-mobile-games

[28] Anon. What is Python? *Opensource* [online]. (nedatováno) [cit. 2024-05-05]. Dostupné z: https://opensource.com/resources/python

[29] SHINNERS, Pete. Python Pygame Introduction. *Pygame* [online]. 2000 [cit. 2024-05-05]. Dostupné z: https://www.pygame.org/docs/tut/PygameIntro.html

[30] Anon. About - pygame wiki. *Pygame* [online]. 2000 [cit. 2024-05-05]. Dostupné z: https://www.pygame.org/wiki/about

[31] Tech With Tim. AI Plays Flappy Bird - NEAT Python. *YouTube* [online]. 2019 [cit. 2024-05-05]. Dostupné z: https://www.youtube.com/playlist?list=PLzMcBGfZo4-lwGZWXz5Qgta_YNX3_vLS2

[32] ANGELOS, Ayla. The history of Snake: How the Nokia game defined a new era for the mobile industry. *It's Nice That* [online]. 2021 [cit. 2024-05-05]. Dostupné z: https://www.itsnicethat.com/features/taneli-armanto-the-history-of-snake-design-legacies-230221

[33] IBM. What is the k-nearest neighbors (KNN) algorithm? *IBM* [online]. (nedatováno) [cit. 2024-05-05]. Dostupné z: https://www.ibm.com/topics/knn

[34] IBM. What is linear regression? *IBM* [online]. (nedatováno) [cit. 2024-05-05]. Dostupné z: https://www.ibm.com/topics/linear-regression

[35] DHUMNE, Shruti. Deep Q-Network (DQN). *Medium* [online]. 2023 [cit. 2024-05-05]. Dostupné z: https://medium.com/@shruti.dhumne/deep-q-network-dqn-90e1a8799871

[36] Anon. Deep Deterministic Policy Gradient. *Spinning Up Open AI* [online]. (nedatováno) [cit. 2024-05-05]. Dostupné z: https://spinningup.openai.com/en/latest/algorithms/ddpg.html

[37] LLC, CodeReclaimers. NEAT Overview. *NEAT-python* [online]. 2019, 2015-2019 [cit. 2024-05-05]. Dostupné z: https://neat-python.readthedocs.io/en/latest/neat_overview.html

[38] freeCodeCamp.org. Python + PyTorch + Pygame Reinforcement Learning – Train an AI to Play Snake. *YouTube* [online]. 2022 [cit. 2024-05-05]. Dostupné z: https://www.youtube.com/watch?v=L8ypSXwyBds&ab_channel=freeCodeCamp.org

[39] Anon. PyTorch documentation. *PyTorch* [online]. 2023 [cit. 2024-05-05]. Dostupné z: https://pytorch.org/docs/stable/index.html

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

AI	Artificial Intelligence.
USA	United States of America.
GPS	General Problem Solver.
MIT	The Massachusetts Institute of Technology
FSM	Finite State Machines
BT	Behavior Trees
DT	Decision Trees
NPC	Non-player Characters
ML	Machine Learning
SVM	Support Vector Machines
KNN	K-nearest neighbor
PCA	Principal Component Analysis
SVD	Singular Value Decomposition
RL	Reinforcement Learning
MDP	Markov decision process
DQN	Deep Q-Networks
DDPQ	Deep Deterministic Policy Gradient
ANN	Artificial neural network
FNN	Feedforward neural networks
CNN	Convolutional neural networks
RNN	Recurrent neural networks
NEAT	NeuroEvolution of Augmenting Topologies
RBS	Rule-based systems
GDPR	General Data Protection Regulation

CCPA California Consumer Privacy Act

SEZNAM OBRÁZKŮ

SEZNAM PŘÍLOH

Elektronická příloha P I:	Složka flappybird – obsahuje zdrojový kód hry Flappy Bird ve formátu Python.
Elektronická příloha P II:	Složka snake – obsahuje zdrojový kód hry Snake ve formátu Python.
Elektronická příloha P III:	Složka obrazky_pdf – obsahuje obrázky ve formátu JPEG, které byly použity v PDF verzi práce.
Elektronická příloha P IV:	Prezentace propag_prehled – obsahuje propagační přehled využívaných metod na hrách Flappy Bird a Snake.
Příloha P V:	Zdrojové soubory na přiloženém CD