STŘEDOŠKOLSKÁ ODBORNÁ ČINNOST

Obor č. 18: Informatika

Implementace Neuronových Sítí v herním prostředí

Ivan Anikin
Hlavní město Praha

Praha 2021

STŘEDOŠKOLSKÁ ODBORNÁ ČINNOST

Obor č. 18: Informatika

Implementace Neuronových Sítí v herním prostředí

Neural Networks implementation in games

Autoři: Ivan Anikin
Škola: Gymnázium Jaroslava Heyrovského, Mezi Školami 2475/29, 155 00 Praha 13-Stodůlky
Kraj: Hlavní město Praha
Konzultant: doc. RNDr. Elena Šikudová, Ph.D.

Praha 2021

Prohlášení

Prohlašuji, že jsem svou práci SOČ vypracoval/a samostatně a použil/a jsem pouze prameny a literaturu uvedené v seznamu bibliografických záznamů.

Prohlašuji, že tištěná verze a elektronická verze soutěžní práce SOČ jsou shodné.

Nemám závažný důvod proti zpřístupňování této práce v souladu se zákonem č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších předpisů.

V Praze dne 03.02.2021 ……
Ivan Anikin

Poděkování

Chtěl bych poděkovat nejprve svému učiteli informatiky Marku Fialovi a jeho kolegyni Lence Giannitsi za nabídnutí možnosti účasti v projektu SOČ a poskytnutí potřebné pomoci. Dále bych chtěl poděkovat Eleně Šikudové (mé konzultantce z MATFYZ-u), která mi pomohla se účastnit hodin Implementace Neuronových Sítí v angličtině na MATFYZ-u a za další pomoc při práci na projektu.

Anotace

Ve své práci SOČ jsem se zabýval využitím neuronových sítí ve 2D herních prostředích na platformě OpenAi Gym. Vytvořil jsem herní Boty, které se při interakci s prostředím učili a zlepšovali se v dosažení určených cílů. Program je psaný v programovacím jazyce Python s využitím knihoven pro strojové učení Tensor Flow a Keras. Modely Neuronových sítí herních Botů byli trénováni pomocí různých metod a docílili výher v různých hrách.

Klíčová slova

Strojové učení; Umělá inteligence; Neuronové Sítě, Počítačové Hry

Annotation

In my SOČ work I was working on the use of neural networks in 2D gaming environments on the platform OpenAi Gym. I created game Bots, which learned and improved in achieving the goals, during interaction with the environment. The program is written in the Python programming language using the Tensor Flow and Keras machine learning libraries. Models of Neural Networks of Game Bots were trained using various methods and won in several games.

Keywords

Machine Learning; Artificial Intelligence, Neural Networks, PC Games

Obsah

1 Úvod

Téma strojového učení mě velice zajímalo už více let, tento rok, po absolvování kurzu na platformě Coursera od Stanfordu na téma Strojového učení, jsem se rozhodl tomuto tématu věnovat nejen teoreticky, ale načerpané znalosti z delšího studia využít prakticky.

Vybral jsem se právě herní prostředí pro implementaci metod strojového učení, protože to bylo pro mě osobně nejzajímavější a nejzábavnější sféra z těch, co mě napadly. Použil jsem základní a podle mě velice dobře propracovanou platformu na vyvíjení Herních Bot-ů OpenAi Gym, která je zaměřena na využití strojového učení.

V této odborné práci se pokusím co nejjasněji popsat princip fungování Strojového Učení, Neuronových sítí a konkrétních metod sbírání dat a trénování bot-ů na jejich základě.

Mým cílem, kterého se mi úspěšně podařilo dosáhnout, bylo vytvoření několika různých druhů modelů, které by dokázali bez využití algoritmů s přesnými podmínkami, jen pomocí samostatného trénování vyhrát (docílit určitého počtu bodů) v takové sadě her, která vhodně pokrývá problematiku nasazení AI.

Všechny odborné pojmy jsou vysvětleny níže v textu.

2 TEORETICKÁ ČÁST

2.1 Umělá inteligence a strojové učení

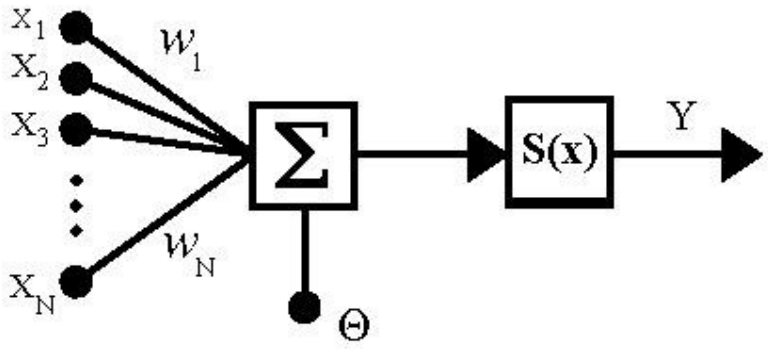
Pojmy AI a strojové učení se stále častěji objevují online i v rozhovorech lidí. Pokusím se je tedy objasnit co nejjasnější a nejpřesnější formou. Pojem umělá inteligence, tak jak ho chápeme teď, se poprvé objevil v roce 1956 v Dartmouth College v Hanoveru, dříve to byla spíše záležitost filosofie než praktického využívání a studování. Dlouhou dobu se tento obor moc nevyvíjel, ale ke konci 20. století začal nabírat obrat a v roce 1997 IBM Deep Blue počítač poprvé vyhrál titul světového šampiona v šachu, když porazil ruského grandmastera Garri Kasparova.

Strojovým učením nazýváme počítačovou vědu, která se zabývá vývojem programového zabezpečení, které se zdokonaluje a učí se řešit úlohu, na kterou je zaměřený pořád přesněji, snižováním odchylky od správného řešení, které může být určené člověkem (supervised learning) a nebo vygenerované z prostředí ve kterém se nachází, či třízení vstupních prvků do skupin podle společných vlastností (unsupervised learning). Strojové učení se dá pojímat jako součást umělé inteligence, ta je spíše obecným pojmem znamenajícím, jakýkoliv uměle vytvořený nástroj pro rozhodování se bez přesně zadaných algoritmů nebo kritérií pro nalezení řešení, pomocí samostatného vývoje.

2.2 Neuronové sítě

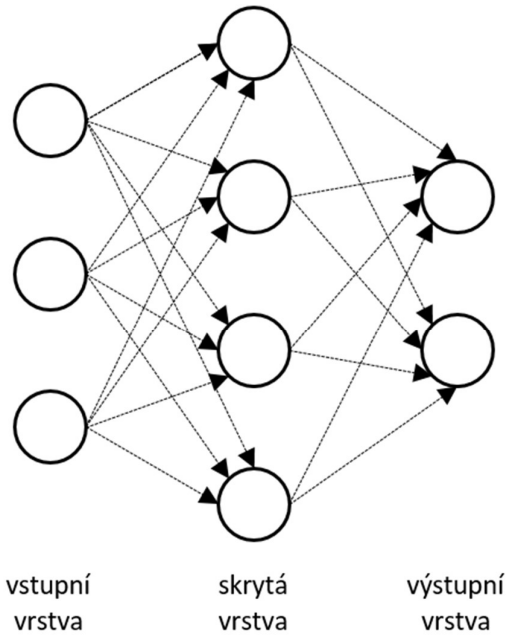
Všechny modely strojového učení použité v této práci byly postaveny na základě takzvaných neuronových sítí. Neuronové sítě jsou určeny pro distribuované paralelní zpracování dat. Skládají se z umělých neuronů, inspirovaných biologickými neurony v lidském mozku, které jsou navzájem různým způsobem propojeny, předávají si signály (ve formě jejich hodnot) a transformují je pomocí různých přenosových funkcí jako jsou například: skoková, hyperbolické tangenty nebo sigmoidální – používaná v tomto projektu).

Neuron v sítí přijímá hodnoty x z předchozích neuronů nebo ze vstupu, násobí je váhami w, které jsou následně měněny během trénování modelu pomocí různých způsobů a posílá výstupní hodnotu (signál) y na další neurony, se kterými je propojen, případně na výstup.



Obr. 1: Model neuronu (Převzato z Umělá neuronová síť – wikipedia.org)

Neuronové sítě použité v tomto projektu jsou takzvaného vrstevnatého druhu. Jsou tedy rozděleny do vrstev a neurony jsou navzájem propojeny, jak je to znázorněno na obrázku, s každým v přechozí a následující vrstvě.



Obr. 2: Model vrstevnaté neuronové sítě (Převzato z Neuronové sítě ... | napocitaci.cz)

3 PRAKTICKÁ ČÁST

3.1 Použité technologie

Z dostupné nabídky nástrojů pro práci s AI, jsem si vybral TensorFlow. Důvodem bylo rozsáhlé množství výstižných zdrojů pro studování a díky tomu, že umožňuje velice příjemnou práci díky vysokoúrovňovým podknihovnám jako například Keras.

3.1.1 TensorFlow

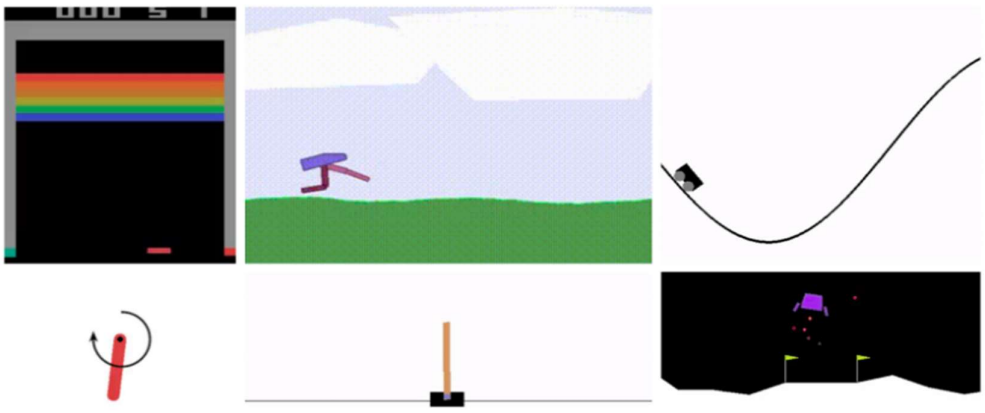
Tensorflow je opensource knihovna vytvořená Google Brain týmem pro numerické výpočty, velkoměřítkové strojové učení a jednoduché vytváření modelů strojového učení.

3.1.2 Keras

Keras je API pro práci s takzvaným hlubokým učením v Pythonu, která běží nad TensorFlowem. Mám velice dobré zkušenosti s prací s touto knihovnou, velice zjednodušuje práci s neuronovými sítěmi, na vytvoření pětivrstvé neuronové sítě stačí použít pět řádků kódu. Doporučoval bych však používat takovýto API vyšší úrovně až po pochopení, toho, co přesně knihovna děláa jak vypadá na nižších úrovních kódu.

3.1.3 OpenAi Gym

OpenAI Gym je sada nástrojů vyvinutá neziskovou výzkumnou společností, která se soustřeďuje na vývoj umělé inteligence. Společnost založil Elon Musk se Samem Altmanem a nabízí různá prostředí pro aplikování algoritmů strojového učení. Z programu pro importování knihovny Gym můžeme vytvořit prostředí (env), ze kterého, po každém kroku, ve kterém provedeme nějakou činnost (většinou pohyb nebo síla pohybu), získáme informace o našem stavu v tomo prostředí, jako je samotný "postřeh" (obrázek jako RGB matice nebo taky jen vektor pohybu), ohodnocení pozice a další.



Obr. 3: Ukázka herních prostředí v OpenAi Gym

Ukázka základního kódu v Python-u pro práci s OpenAi Gym prostředím:

```
import gym

episodes = 10
sim_steps = 500


env = gym.make("CartPole-v0")



# loop through several episodes
for ep in range(episodes):
     # resetting the environement is neccesary to start a new episode
     env.reset()

     # one episode loop trough steps count
     for step in range(sim_steps):

          # random action from action space
          action = env.action_space.sample()

          # receiveing output from environement after taking action
          observation, reward, done, info = env.step(action)

          # end episode loop if break == true (one of env outputs)
          if done:
               break;
```

3.2 Struktura zdrojového kódu

Kód pro adaptivnost, lehkost v používání a přehlednost mám rozdělený do souborů a tříd.

3.2.1 Agenti

V této kategorii se nachází soubory, které jsou využívány programy zaměřenými na konkrétní úkol.

Agent

Tento soubor v sobě obsahuje různé agenty strojového učení – třídy které zahrnují proměnné, jako jsou parametry modelu, samotný model, statistiky během tréninku a další. Obsahuje také funkce, základními jsou například: Learn(trénování modelu) nebo Run(vrácení hodnoty výstupu modelu pro konkrétní vstup). Samotní agenti, jako třídy, jsou pojmenováni podle obecného zařazení modelu, jako je Q-Learning, DQL(Deep Q-Learning), Actor-Critic a další.

Models

V souboru Models najdeme třídy, které obsahují proměnné modelů, podle jejich typu. Actor-Critic například obsahuje jako DQL model dva modely nebo jeden s dvěma výstupy. Třída Agent z tohoto souboru získává data pro tvorbu modelů, kterými se trénuje.

Wrapper

Jako další výpomocný soubor bych chtěl zmínit Wrapper, v něm se nachází třídy a funkce, které modifikují stav, který agent z prostředí získává (místo [255;255;3] RGB matice jako herní obrazovku vrací [84;84;3] stav hry s komprimovaným pohledem pro zrychlení tréninku). Modifikuje také ocenění, které agent od prostředí získává, a podle kterého se následně učí, jak dobré rozhodnutí, jež udělal, bylo.

Další

Jinými soubory pro použití z různých části kódu jsou například agenti na práci s úložištěm, na vizualizaci výsledků a statistik trénování nebo hyperparametry, jako výchozí proměnné odpovídající více různým prostředím.

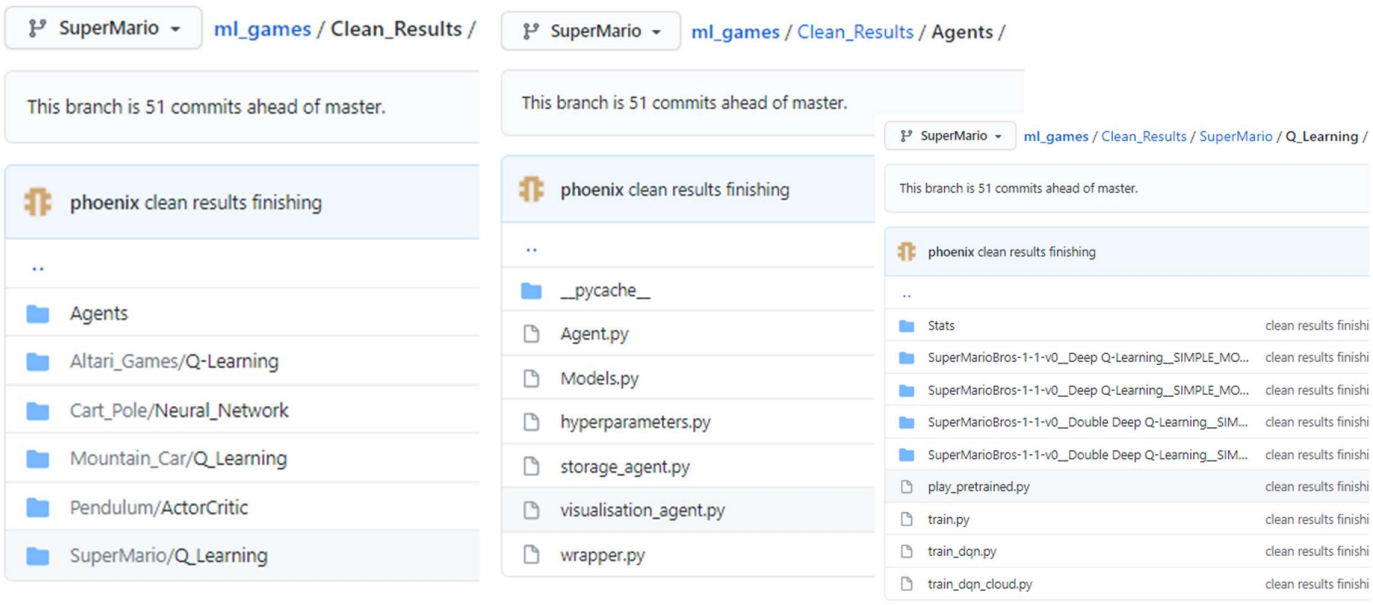
3.2.2 Složky s názvy podle konkrétní úlohy

Z důvodu různorodosti úloh (vstupních informace a potenciálních možností v prostředí reagovat) se nedá udělat něco na způsob General AI, jež by mohl univerzálně řešit všechny úlohy. Něco takového by bylo neefektivní v každém z případů.

Proto jsem vytvořil specifický způsob trénování (model, algoritmus trénování modelu a další specifické části kódu) pro každou úlohu zvlášť. Výjimkou je složka Atari_Games, ve které jsou algoritmy trénování modelů využitelné pro více her typu ATARI, protože vstupem získávaným z prostředí jsou matice obrazovky ve stejném formátu a rozhodnutí činu agenta jsou vždy na výběr z předem určeného počtu možností.

V těchto složkách se nachází soubor Train, nebo několik takových, pokud obsahují více možností trénování modelů, v každém takovém souboru jsou zadány parametry trénování a stavby modelu, což umožňuje specifické možnosti trénování.

Dále se v těchto složkách nachází natrénované modely a soubory se statistikami z přechozích tréninků pro příklad.



3.3 Vysvětlení samotného trénování modelů

Nejprve jsem řešil několik základních úloh v OpenAi Gym. Každá z nich by však bez využití strojového učení byla pokud ne neřešitelná, tak mnohem náročnější. Na těchto úlohách jsem vyzkoušel různé modely a metody jejich trénování, jejich výhody i nevýhody, a pro jaké příklady jsou určeny. Poté jsem se pustil do ATARI her, začínajíce SuperMario Bros, které pro mě byly cílovými.

V každé úloze se setkáte s pojmem ohodnocení, který často využívám. Pro objasnění, tímto pojmem je myšleno ohodnocení od prostředí Gym pozice v každém kroku hry, podle toho za jak úspěšné je statisticky považována. (vyšší ocenění při blízkosti vítězství, získání extra bodů atp.)

3.3.1 Supervised Learning – CartPole úloha

Cílem této úlohy je udržet nestabilní tyč na vozíku na vodorovné podložce ve svislé pozici nakláněním podložky vlevo nebo vpravo. Každý krok získáváme určitá vstupní nebinární data (ne 0 a 1) a jako výstup máme na výběr ze dvou možností, náklon doprava nebo doleva. Cílem je udržet tyč co nejdéle na místě.

Tato forma trénování je založena na datasetu, ve kterém jsou vytýčené správné a špatné příklady pro trénování.

Obr. 4: Náhled na úlohu CartPole

Tento způsob jsem byl schopen použít jen v této nejlehčí úloze CartPole, ve které se můj kód dá rozdělit do tří částí:

1. sbírání dat jako dobrých příkladů,
2. trénování modelu,
3. testování úspěšností natrénovaného modelu.

Sbírání dat bylo prováděno tímto způsobem:

```
    scores = []
    num_trials = 10000
    sim_steps = 500
    min_score = 70

    for episode in range(num_trials):
        observation = env.reset()
        score = 0
        training_sampleX, training_sampleY = [], []
        for step in range(sim_steps):
            # taking random action
            action = np.random.randint(0, 2)
            # converting action into one_hot action array - [0, 1]
            one_hot_action = np.zeros(2)
            one_hot_action[action] = 1
            training_sampleX.append(observation)
            training_sampleY.append(one_hot_action)
            # getting env output after taking aciton
            observation, reward, done, _ = env.step(action)
            score += reward
            if done:
                break
        # appending training X and trianing Y for training the model as a
good example, beacause it has score > than minimal set score
        if score > min_score:
            scores.append(score)
            trainingX += training_sampleX
            trainingY += training_sampleY
```

TrainingX je nyní matice stavů (observation) a TrainingY je matice kroků, které jsme v tu chvíli udělali. Slouží jako příklad vstupní a výstupní vrstvy modelu, který backpropagací obnoví váhy *w* neuronové sítě, tak aby odpovídali co nejvíce konverzi X do Y.

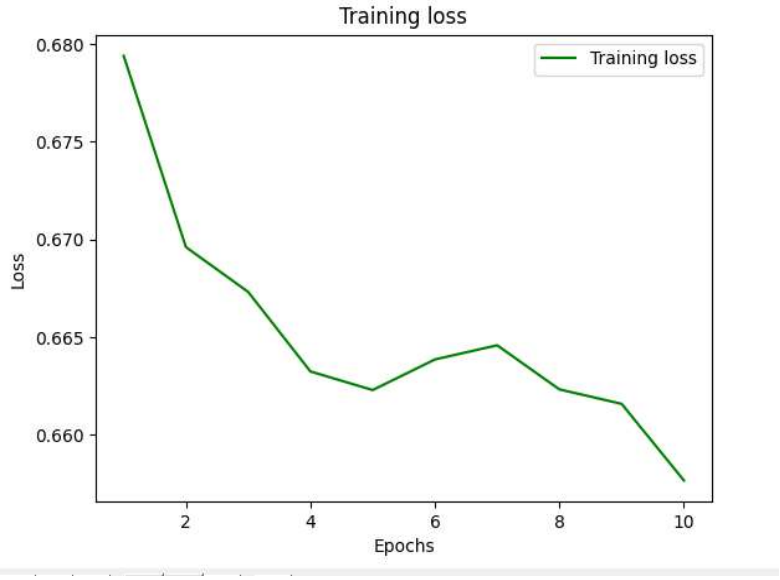
Pomocná funkce create_model vrací vytvořený model ve formátu Keras knihovny. Tento konkrétní model se skládá z: jedné vstupní vrstvy s počtem neuronů odpovídajícím velikosti matice observation, kterou získáváme z prostředí, následují vnitřní vrstvy s počtem neuronů 128, 256, 512, 256, 128 (tyto hodnoty nejsou náhodné, ze statistik vyplývá, že násobky 128 mají lepší výsledky přesnosti trénování) a následuje výstupní vrstva neuronů, skládající se ze dvou neuronů, což odpovídá počtu možných činností které může agent v prostředí vykonat.

Zkoušel jsem více různých způsobů vypočítávání odchylky od správného rozhodnutí a samotného obnovování vah neuronů a pro konkrétní příklady tohoto druhu (s binárními rozhodnutími a číselnými vstupy) se mi nejvíce osvědčila metoda "categorical cross entropy" a optimizer Adam. Pro více informací o optimizerech a vypočítávání odchylky doporučuji oficiální stránky keras-u: Losses (keras.io), Optimizers (keras.io).

```
def create_model(LR, dropout):
    model = Sequential()
    model.add(Dense(128, input_shape=(4,), activation="relu"))
    model.add(Dropout(dropout))

    model.add(Dense(256, activation="relu"))
    model.add(Dropout(dropout))

    model.add(Dense(512, activation="relu"))
    model.add(Dropout(dropout))

    model.add(Dense(256, activation="relu"))
    model.add(Dropout(dropout))

    model.add(Dense(128, activation="relu"))
    model.add(Dropout(dropout))
    model.add(Dense(2, activation="softmax"))

    model.compile(
        loss="categorical_crossentropy",
        optimizer=keras.optimizers.Adam(lr=LR), # LR = 1e-3
        metrics=["accuracy"])
    return model
```

Po vytvoření neuronové sítě je na čase ji natrénovat na získaných správných příkladech. Obnovování vah v tomto příkladě probíhá v deseti krocích typem optimalizace, learning ratem a počítáním odchylky od hodnot zadaných dříve při tvorbě modelu.

```
epochs = 10
history = model.fit(trainingX, trainingY, epochs=epochs)
```



Obr. 5: Klesání odchylky trénování

Výsledné střední skóre natrénovaných modelů podle počtu trénovacích příkladů:

Náhodné rozhodování: 20 2.500 trénovacích příkladů: 120

5.000 trénovacích příkladů: 190 10.000 trénovacích příkladů: 240

Tento způsob by šel vylepšit přidáváním příkladů do správných příkladů po překonání středního počtu dosaženého skóre modelem, kterým by sbíral data, podle nich trénoval a přijímal by rozhodnutí. To by se jednalo o Reinforecement Learning, kdy by agent přijímal rozhodnutí v prostředí a trénoval podle úspěšnosti jednotlivých kroků najednou.

Ve všech ostatních příkladech se bude jednat o takzvané Unsupervised Learning, protože model budeme trénovat bez předem určených správných řešení.

3.3.2 Q.Learning - MountainCar úloha

V této úloze ovládáme autíčko ve dvourozměrném prostředí, skládajícího se ze dvou vrcholů na stejné dráze. Cílem této úlohy je vyjet na pravý vrchol. Na výběr máme ze 3 možností: pohyb vlevo, vpravo a žádný. Na dosažení cíle pohybujíce se jen vpravo však bohužel nemá auto sílu, proto je jeho úkolem zhoupnout se na levém kopci. Vstupem je opět sada nebinárních hodnot.

Obr. 6: Náhled na úlohu MountainCar

Q-Learning je metoda založená na hodnotách Q, které přiřazujeme jednotlivým rozhodnutím v konkrétní situaci (při specifickém vstupu). Jelikož v této úloze máme nebinární vstup a chceme řešit úlohu pomocí metody Q-Learning, potřebujeme vstupní data rozdělit ze spojitého spektra všech možností na určitý počet částí (v tomto příkladě, jak se dočtete níže, se mi osvědčilo 20).

To znamená, že data které jsou menší než 1/20 budou patřit do prvního políčka matice, 1/20 -> 2/20 do druhého atd. Tímto způsobem z každého vstupu dostaneme matici o 20 prvcích ze kterých vytvoříme tabulku Q hodnot o rozměrech [počet vstupních hodnot]*[počet částí na které je chceme rozdělit]*[počet výstupů], která nabývá rozměrů 1*20*3. Takto pro každou kombinaci vstupních dat máme tři hodnoty Q, které slouží jako ukazatel, jak dobré je konkrétní rozhodnutí v daný moment.

Následně mým úkolem bylo v cyklu přijímat náhodná rozhodnutí a postupně je zaměňovat rozhodnutími z tabulky Q a obnovovat hodnoty Q, podle toho jaké ohodnocení od prostředí získám v příštím stavu.

Samotné obnovení hodnoty Q se počítá jako součet současné hodnoty*(1-LR) a LR*(ohodnocení po vykonání kroku * DISCOUNT * maximální hodnota Q při novém vstupu)

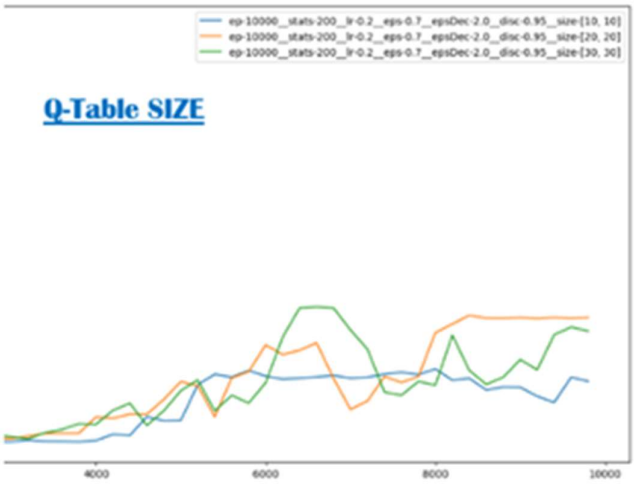
```
new_q = (1 - LEARNING_RATE) * current_q + LEARNING_RATE * (reward +
DISCOUNT * max_future_q)
```

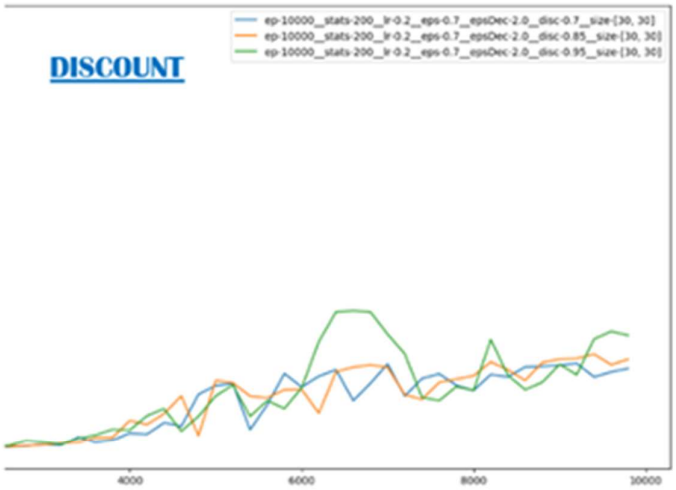
LR udává s jakou rychlostí budeme obnovovat tabulku novýma hodnotami = v tomto příkladě 0.1

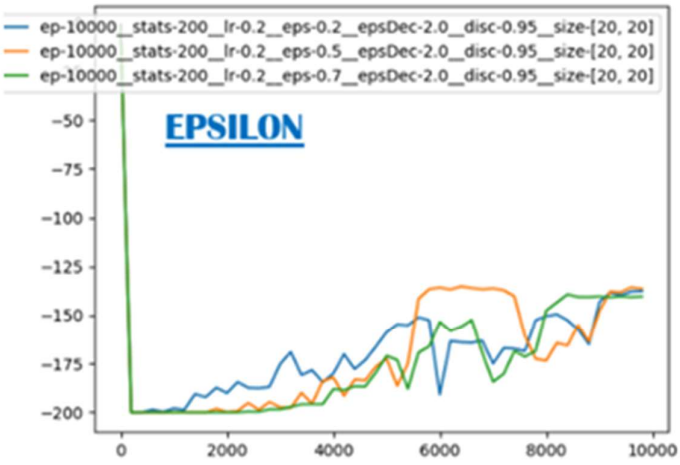
DISCOUNT udává jak velkou hodnotu dáváme příští hodnotě Q. Něco kolem 0.95

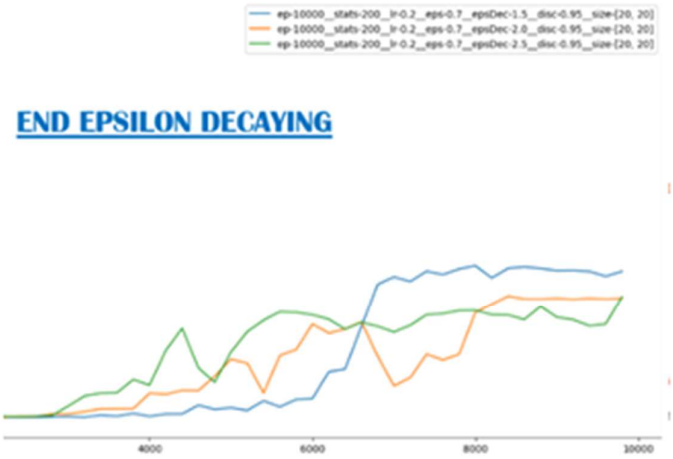
Proměnná epsilon slouží k náhodnému rozhodování a epsilon decay proměnná udává snižování podílu náhodných rozhodnutí a přijatých modelem.

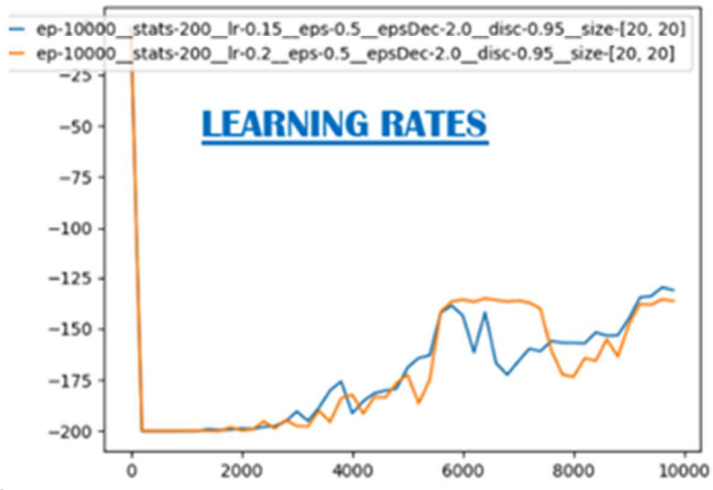
```
    #discrete_os_win_size je hodnota rozdílu mezi jednotlivými díly tabůlky

    DISCRETE_OS_SIZE = [20, 20] # rozměr tabůlky

    q_table = np.random.uniform(low=-2, high=0,
                                size=(DISCRETE_OS_SIZE +
[env.action_space.n]))

    def get_discrete_state(state):
        discrete_state = (state - env.observation_space.low) /
discrete_os_win_size
        return tuple(discrete_state.astype(
            np.int))  # we use this tuple to look up the 3 Q values for
the available actions in the q-table

    for episode in range(EPISODES):
        discrete_state = get_discrete_state(env.reset())

        while not done:

            if np.random.random() > epsilon:
                # Get action from Q table
                action = np.argmax(q_table[discrete_state])
            else:
                # Get random action
                action = np.random.randint(0, env.action_space.n)

            new_state, reward, done, _ = env.step(action)
            episode_reward += reward

            new_discrete_state = get_discrete_state(new_state)

            if not done:

                # Maximum possible Q value in next step (for new state)
                max_future_q = np.max(q_table[new_discrete_state])

                # Current Q value (for current state and performed
action)
                current_q = q_table[discrete_state + (action,)]

                # Equation for a new Q value for current state and action
                new_q = (1 - LEARNING_RATE) * current_q + LEARNING_RATE *
(reward + DISCOUNT * max_future_q)

                # Update Q table with new Q value
                q_table[discrete_state + (action,)] = new_q

            # Simulation ended (for any reson) - if goal position is
achived - update Q value with reward directly
            elif new_state[0] >= env.goal_position:
                q_table[discrete_state + (action,)] = 0

                successful_episodes += 1

            discrete_state = new_discrete_state
```











Obr. 7: Statistiky vývoje středního skóre v závislosti na parametrech

3.3.3 Actor-Critic – Pendulum úloha

Cílem této úlohy je udržet tyč vertikálně vzpřímenou nahoru balancováním určitou silou doleva nebo doprava. Je také specifická v tom, že rozhodnutí, které můžeme přijmout je na rozdíl od předchozích úloh spektrální, není tedy tvořeno volbou z ohraničeného množství činů, ale je to hodnota v tomto případě od -2 do 2 udávající sílu pohybu a znaménkem jeho směr. Právě proto, pro řešení této úlohy není možné použít metody Q-Learning pomocí tabulky, protože máme nekonečné množství rozhodnutí.

Obr. 8: Náhled na úlohu Pendulum

Na tomto příkladě ukážu implementaci metody Actor-Critic. Model je tvořen čtyřmi neuronovými sítěmi, takzvaným hráčem, kritikem a jejich target verzemi. Neuronové sítě herce přijímají na vstupu stav prostředí a jejich výstupem je hodnota rozhodnutí, sítě kritika mají jako vstup stav a tato hodnota rozhodnutí, výstupem je něco na způsob hodnoty Q z předchozího příkladu, která udává, jak dobré to rozhodnutí bylo podle této sítě. Při této metodě neuronová síť herce přijímá rozhodnutí, následně při trénování modelu se obnovuje síť kritika na základě hodnoty target kritika, pro současný stav a přijaté rozhodnutí sítí target herce, a ohodnocení, které bylo získáno po tomuto rozhodnutí. Síť herce se obnovuje tak, že míru správnosti výsledné hodnoty, ke které by se síť měla snažit, udává hodnota kritika pro současný stav a vykonané rozhodnutí. Takto trénujeme nejprve kritika pomocí ohodnocení následujícího po rozhodnutí a následně herce, na základě jak ohodnotí jeho přijaté rozhodnutí kritik. Po každém trénovacím kroku se obnovují váhy target sítí na váhy sítí herce a kritika.

Actor-Critic je možné také vytvořit pomocí jen dvou, nebo i jedné sítě, která bude obnovovat rovnou sebe stejným způsobem na základě ohodnocení od prostředí. Bude mít stejný vstup stavu a přijatého rozhodnutí a dva výstupy: hodnota rozhodnutí a kritik hodnoty ohodnocení. Výásledek tohoto postupu však není tak přesný jako rozdělení do více sítí a chtěl jsem v tomto příkladě zahrnout i target modely. Tím, že trénujeme model podle hodnot target modelů a až poté ho obnovujeme, nám umožnuje se vyvarovat trénování na anomálních příkladech z jednoho tréninku.

Tvar neuronových sítí vypadá takto:

Herec: [počet vstupních hodnot] | 256 | 256 | [počet rozhodnutí]

Kritik: [počet vstupních hodnot] | 16 | 32 |

[počet rozhodnutí] | 32 |

256 | 256 | 1

Některé části kódu použitého pro tuto úlohu byly použity z opensource zdroje dokumentace Kerasu: [keras-team/keras-io: Keras documentation, hosted live at keras.io (github.com)]

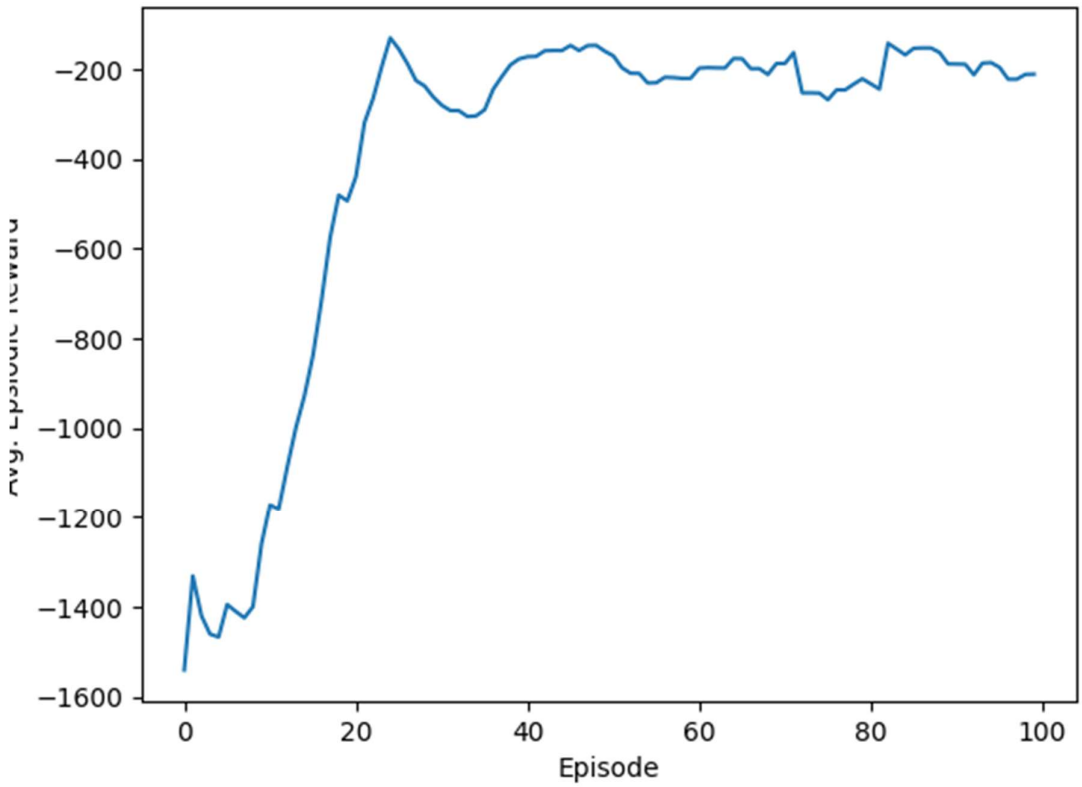
Takto vypadá jeden krok trénování modelu:

```
def update(self, state_batch, action_batch, reward_batch,
next_state_batch,):
    # Training and updating Actor & Critic networks.
    # See Pseudo Code.
    with tf.GradientTape() as tape:
        target_actions = self.models.target_actor(next_state_batch,
training=True)
        y = reward_batch + self.gamma * self.models.target_critic(
            [next_state_batch, target_actions], training=True
        )
        critic_value = self.models.critic_model([state_batch,
action_batch], training=True)
        critic_loss = tf.math.reduce_mean(tf.math.square(y -
critic_value))

    self.critic_grad = tape.gradient(critic_loss,
self.models.critic_model.trainable_variables)
    self.models.critic_optimizer.apply_gradients(
        zip(self.critic_grad,
self.models.critic_model.trainable_variables)
    )

    with tf.GradientTape() as tape:
        actions = self.models.actor_model(state_batch, training=True)
        critic_value = self.models.critic_model([state_batch, actions],
training=True)

        actor_loss = -tf.math.reduce_mean(critic_value)

    actor_grad = tape.gradient(actor_loss,
self.models.actor_model.trainable_variables)
    self.models.actor_optimizer.apply_gradients(
        zip(actor_grad, self.models.actor_model.trainable_variables)
    )

    self.update_target(self.models.target_actor.variables,
self.models.actor_model.variables, self.tau)
    self.update_target(self.models.target_critic.variables,
self.models.critic_model.variables, self.tau)

    @tf.function
    def update_target(self, target_weights, weights, tau):
        for (a, b) in zip(target_weights, weights):
            a.assign(b * tau + a * (1 - tau))
```

Střední dosažené skóre po různém počtu epizod trénování:

Náhodně: -1600 Epizoda 25: -300 Epizoda 50: -250

Epizoda 75: -200 Epizoda 100: -180



Obr. 9: Zvyšování středního skóre s počtem epizod trénování

3.4 Aplikace získaných poznatků

Mým hlavním cílem bylo napsat algoritmus strojového učení, který by dokázal vyhrát ve hře SuperMario, práci mi velice zjednodušilo rozšíření ATARI her, které nabízel OpenAi Gym, které zahrnovalo jak SuperMario Bros, tak mnoho dalších, do kterých jsem se také pustil. Na předchozích úlohách jsem si natrénoval různé metody trénování a rozhodl jsem se pustit do ATARI prostředí, které bylo náročnější kvůli vstupu ve formě snímku obrazovky.

3.4.1 Deep Q-Learning – SuperMario a ATARI Games

Super Mario hra a jiné ATARI hry v prostředí OpenAi Gym jsou stejné v tom, že jako vstup získáváme RGB matici [255;255] jako snímek obrazovky a výstupem je určitý počet tahů, ze kterých si vybíráme vždy jeden. To nám umožnuje napsat jeden algoritmus pro vyřešení všech úloh pomocí jednoho algoritmu. Naším modelem v tomto příkladě je neuronová síť, která získává na vstup konvertovaný vstup tří kroků obrazovky, tedy 84·84·3 matici a výstupem jsou hodnoty Q pro každou z možností rozhodnutí v dané situaci. Jako v bodě 3.3.3, i v této úloze jsem použil druhou target neuronovou síť, kterou obnovujeme po určitém počtu kroků.

Tady je ukázka toho, jakým způsobem je prováděn trénovací krok, kdy v závislosti na tom, jestli používáme metodu Double Deep QL počítáme správný výstup pro daný stav jiným způsobem.

```
        next_q = self.model_target(next_state)
        # Calculate discounted future reward
        if self.double_q:
            q = self.model_online(next_state)
            a = np.argmax(q, axis=1)

            target_q = [0]*self.batch_size

            index = 0
            for TQ in target_q:
                one_target_q = reward[index] + (1. - done[index]) *
                                self.gamma * next_q[index, a[index]]
                target_q[index] = one_target_q
                index+=1
        else:
            target_q = reward + (1. - done) * self.gamma *
                          np.amax(next_q, axis=1)


        self.a_true = np.array(action)
        self.q_true = np.array(target_q)
        current_states_q_values = self.model_online(state)

        X = state
        Y = np.array(current_states_q_values)

        index = 0
        for one_q_true in self.q_true:

            Y[index, self.a_true[index]] = one_q_true

            index+=1

        self.model_online.fit(X, Y, verbose=0)
```

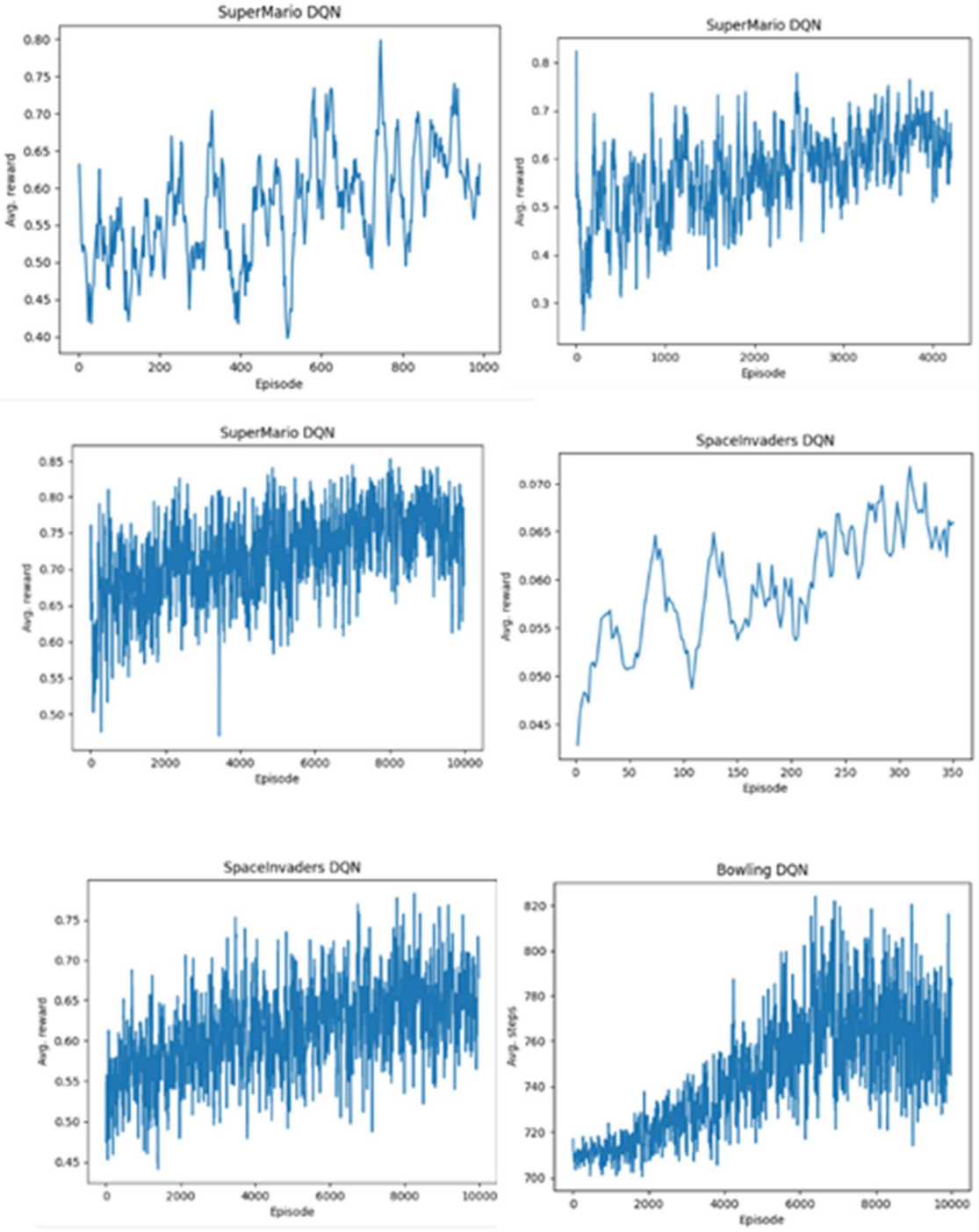
V mém případě měl Mário na výběr ze sedmi možností co v každém kroku udělat:

```
SIMPLE_MOVEMENT
= [
                    ['NOOP'],
                    ['right'],
                    ['right', 'A'],
                    ['right', 'B'],
                    ['right', 'A', 'B'],
                    ['A'],
                    ['left'],
                ]
```

Cílem jak to známe z běžné hry je dosáhnout vlajky na konci úrovně.

Úspěšně se mi podařilo vytvořit trénovací algoritmus s metodou Deep Q-Learning, který je schopný vyřešit několik desítek různých ATARI her, aniž by o nich cokoliv věděl. Trénování probíhá jen na základě ohodnocení, které od prostředí získáváme (za posun dále doprava, sbírání bodíků atd.). Takovýto způsob trénování, aplikovatelný pro vícero rozdílných úloh se dá považovat za nástřel ve směru GAI – Global Artificial Intelligence, umělá inteligence využitelná pro více rozlišných úloh. V každé z těchto úloh však získáváme stejný vstup jako snímek obrazovky a výstup se shoduje v tom, že se jedná o výběr jedné možnosti a ne o hodnotu rozhodnutí na spektru.

Progresi učení AI pro konkrétní hry SuperMario, SpaceInvaders a Bowling znázorňují následující grafy. Uvedené hry považuji za tak známé, že upouštím od záměru je čtenáři představovat.



Obr. 10:Vývoj středního ohodnocení v různých ATARI

4 ZÁVĚR

V této práci jste mohli vidět dosažené výsledky v různých úlohách v herním prostředí úspěšně vyřešené pomocí modelů strojového učení. Byly zde také uvedeny statistiky trénování v závislosti na různých parametrech.

Stejné způsoby trénování by se dali potenciálně využít pro automatizaci fyzických robotů, jako by mohlo být autíčko, robotická ruka, nebo jiní roboti, které by při interakcí s prostředím se správně nastavenými hodnoceními, podle toho, co by bylo jejich úlohou, mohli naučit plnit jejich účel bez psaní algoritmů předpovídající všechny možnosti. Zrovna do ovládání autíčka mám v plánu se pustit hned po tomto projektu, jeho cíle by mohly být nalezení objektů v prostředí ve kterém se nachází, nebo jednodušší možností by bylo jízda za světelným signálem. Jako vstup mi přijde ideální použít kameru, nebo dvě pro rozpoznávání vzdáleností. První částí tohoto projektu bude ho naučit jezdit a prozkoumávat okolní prostředí bez narážení.

Praktické využití následně natrénovaných robotů by mohlo být vyhledávání objektů v prostředí, transport předmětů, nebo i výpomoc s úklidem.

Jednotlivé typy bibliografických záznamů se nedělí do více skupin (např. Internetové zdroje, Články, Knihy…), ale uvádějí v jednom neděleném seznamu.

Zdrojový kód všech úloh se nachází na tomto úložišti na GitHubu. Doufám, že zájemci nebudou mít problém s orientací ve zdrojovém kódu pro další využití, či jen ozkoušení.

ml_games/Clean_Results at master · IvanAnikin/ml_games (github.com)

5 POUŽITÁ LITERATURA

[1] Machine Learning by Stanford University | Coursera

[2] Machine learning - Wikipedia

[3] Optimizers (keras.io)

6 SEZNAM OBRÁZKŮ A TABULEK