VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
PROGRAMŲ SISTEMOS

Skaitmeninis intelektas ir sprendimų priėmimas
Žaidimas „Kryžiukai–nuliukai“

Darbą atliko:
Vadim Čeremisinov
Pijus Zlatkus

Vilnius
2024

Turinys

1. Užduoties tikslas

Užduoties tikslas – sukurti programą, įgyvendinančią žaidimą „Kryžiukai-nuliukai“, taikant daugiasluoksnį tiesioginio sklidimo neuroninį tinklą.

2. Žaidimo taisyklės ir modelio sprendžiamas uždavinys

Tai yra paprastas, nesunkus ir dviem žaidėjams skirtas žaidimas. Žaidimo laukas: 9 langelių (3×3) lentelė. Vienas žaidėjas lentelės langelius žymi X (kryžiukais), antrasis – 0 (nuliukais). Žaidėjai eina paeiliui. Laimi tas, kuris pirmas užpildo pilną eilutę, stulpelį ar įstrižainę savo ženklais.

Išmokyti neuroninį tinklą žaisti šį žaidimą yra sprendžiamas **klasifikavimo uždavinys**. Neuroninis tinklas turi nuspręsti, koks veiksmas (t.y., kur padėti nuliuką ar kryžiuką) yra geriausias esamoje žaidimo būsenoje. Tai galima laikyti klasifikavimo problema, nes iš visų galimų veiksmų (paprastai nuo 1 iki 9, priklausomai nuo tuščių langelių skaičiaus) reikia pasirinkti vieną.

3. Žaidimo neuroninio tinklo realizacija

3.1. Naudojami programiniai įrankiai ir resursai

Neuroninio tinklo realizacijai buvo pasirinkta naudoti Python programavimo kalbą su neuroniniams tinklams skirta biblioteka TensorFlow. Ši biblioteka ne tik suteikia reikalingą funkcionalumą kurti ir mokinti neuroninius tinklus, bet ir prisitaikyti prie techninės įrangos ir skaičiavimo resursų. Mokinti neuroninio tinklo modeliui reikia nemažai skaičiavimo resursų, todėl naudoti kompiuterio procesorių šiam tikslui nėra efektyvu ir tai gali užtrukti daug laiko. Geriausia tai atlikti pasitelkiant kompiuterio grafinį procesorių (GPU). Tai leido vykdyti visą mokymo procesą asmeniniuose kompiuteriuose.

3.2. Neuroninio tinklo architektūra

Šiam uždaviniui atlikti buvo naudojamas daugiasluoksnis tiesioginio sklidimo neuroninis tinklas. Kadangi žaidimas yra paprastas, tam nereikėjo turėti sudėtingą architektūrą. Geriausią veikimą parodė architektūra parodyta kode esančiame **1 pav.**.

```python
# Neuroninio tinklo modelio apibrėžimas
def create_model():
    # Sukuriame sekos modelį
    model = Sequential([
        # "Flatten" sluoksnis ištiesina 3x3 įėjimo matricą į 9 elementų vektorių
        Flatten(input_shape=(3, 3)),
        # Pirmasis tankus (Dense) sluoksnis su 128 neuronais ir "relu" aktyvacijos funkcija
        Dense(36, activation='relu'),
        # Antrasis tankus sluoksnis su 128 neuronais ir "relu" aktyvacijos funkcija
        Dense(36, activation='relu'),
        # Trečiasis tankus sluoksnis su 9 neuronais (po vieną kiekvienam galimam veiksmui) ir "linear" aktyvacijos funkcija
        Dense(9, activation='linear')
    ])
    # Kompiliuojame modelį, naudojant "adam" optimizatorių ir "mse" nuostolių funkciją
    model.compile(optimizer=Adam(learning_rate=0.1), loss='mse')
    # Grąžiname sukurtą modelį
    return model
```

1 pav. *Neuroninio tinklo modelio architektūra*

Pateiktas daugiasluoksnis tiesioginio sklidimo neuroninio tinklo modelis buvo sukurtas naudojant *Sequential* klasę iš Keras bibliotekos. Modelis susideda iš trijų sluoksnių: pirmasis sluoksnis yra *Flatten* sluoksnis, kuris paverčia 3x3 įėjimo matricą į 9 elementų vektorių, antrasis sluoksnis yra pilnai jungus (angl. fully-connected layer) sluoksnis su 36 neuronais ir *ReLU* (angl. Rectified Linear Unit) aktyvacijos funkcija, o trečiasis sluoksnis taip pat yra pilnai jungus sluoksnis su 36 neuronais ir ReLU aktyvacijos funkcija. Paskutinis sluoksnis yra išvesties sluoksnis su 9 neuronais ir tiesine aktyvacijos funkcija.

Modeliui buvo pritaikytas *Adam* (angl. Adaptive Moment Estimation) optimizavimo metodas ir *MSE* (angl. Mean Square Error) nuostolių funkcija, su nustatytu mokymosi greičiu lygiu *0,1*. Šis modelis parodė geriausius rezultatus paprastam žaidimui, kuriam nereikėjo sudėtingos architektūros, ir buvo naudojamas tolimesniems tyrimams bei eksperimentams.

3.3. Apmokymo strategija

Apmokymo strategija žaidimo kryžiukų-nuliukų neuroniniam tinklui buvo paremta savarankiško mokymosi metodu, kai modelis treniruojasi žaisdamas prieš save patį, naudojant agentą, kuris prižiūri mokymo procesą. Šiam atvejui buvo būtent panaudotas Q-learning algoritmas.

3.3.1. Q-learning algoritmas

Q-learning yra skatinamojo mokymosi (angl. reinforcement learning) algoritmas, naudojamas mokytis siekiant pasiekti didžiausią atlygį.

Algoritmo veikimas:

1. **Q-Vertės:** Q-learning algoritmas palaiko Q-vertės (kokybės vertės) funkciją $Q(s, a)$, kuri atspindi tikėtiną sukauptą atlygį atliekant veiksmą a būsenoje s.
2. **Veiksmų Pasirinkimas:** Agentas renkasi veiksmus remdamasis Q-vertėmis. Dažnai naudojama ϵ-godumo strategiją, kuri parinka vieną iš šių veiksmų:
    a. Su tikimybe ϵ agentas pasirinks atsitiktinį veiksmą (tyrinėjimas).
    b. Su tikimybe $1 - \epsilon$ agentas pasirinks veiksmą, turintį aukščiausią Q-vertę.
3. **Q-Funkcijos Atnaujinimas:** Po kiekvieno veiksmo agentas atnaujina Q-vertes pagal gautą atlygį ir būsimos būsenos vertę. Atnaujinimas vykdomas pagal formulę:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q\,(s', a') - Q(s, a) \right]$$

3.3.2. Duomenų generavimas

Kadangi kryžiukų-nuliukų žaidimas yra paprastas ir lengvai simuliuojamas, mokymo duomenys buvo generuojami dinamiškai, modeliams žaidžiant tarpusavyje. Tai leido išvengti poreikio turėti iš anksto paruoštą didelį duomenų rinkinį.

3.3.3. Aplinka

Aplinka buvo sukurta naudojant *TicTacToe* klasę, rodomą **2 pav.**, kuri simuliuoja žaidimo lentą ir taisykles. Ši klasė leidžia atlikti veiksmus, tikrina jų validumą ir nustato, kada žaidimas baigiasi, grąžindama atitinkamus atlygius.

```python
# Kryžiukų-nuliukų žaidimo logikos apibrėžimas
class TicTacToe:
    def __init__(self):
        # Inicijuojame žaidimo lentą kaip 3x3 matricos nulių masyvą, kur 0: tuščias langelis, 1: X, 2: O
        self.board = np.zeros((3, 3), dtype=int)
        # Inicijuojame dabartinį žaidėją kaip 1 (X)
        self.current_player = 1  # 1: X, 2: O

    def reset(self):
        # Atstatome žaidimo lentą į pradinę būseną (visi langeliai tušti)
        self.board = np.zeros((3, 3), dtype=int)
        # Nustatome dabartinį žaidėją į 1 (X)
        self.current_player = 1
        # Grąžiname atstatytą lentą
        return self.board

    def step(self, action):
        # Patikriname, ar veiksmas galimas (langelis turi būti tuščias)
        if self.board[action] != 0:
            raise ValueError(f"Invalid action: {action}")

        # Atliekame veiksmą - įrašome dabartinį žaidėją į pasirinktą langelį
        self.board[action] = self.current_player
        # Patikriname žaidimo būseną po veiksmo
        reward, done = self.check_game_status()
        # Pakeičiame dabartinį žaidėją (3 - 1 = 2; 3 - 2 = 1)
        self.current_player = 3 - self.current_player
        # Grąžiname atnaujintą lentą, atlygį ir žaidimo pabaigos flagą
        return self.board, reward, done

    def check_game_status(self):
        # Patikriname, ar kuris nors žaidėjas laimėjo pagal eilutes ir stulpelius
        for i in range(3):
            if all(self.board[i, :] == self.current_player) or all(
                self.board[:, i] == self.current_player
            ):
                return 1, True  # Pergalė

        # Patikriname, ar kuris nors žaidėjas laimėjo pagal įstrižaines
        if (
            self.board[0, 0]
```

2 pav. *TicTacToe klasės struktūra*

3.3.4. Modelio mokymas

Modelio treniravimo procesas buvo įgyvendintas naudojant savarankiško žaidimo, **3 pav.** esančią funkciją *self_play*, kur modelis treniruojasi žaisdamas prieš save. Treniravimo metu modelis naudojo Q-learning algoritmą, kurio metu agentas (modelis) mokosi optimalios strategijos, atnaujindamas savo žinias po kiekvieno žingsnio.

```python
# Mokymo ir vertinimo funkcijos
def self_play(agent, env, episodes, metrics):
    for episode in range(episodes):
        # Spaudiname einamojo epizodo numerį
        print(f"Episode {episode + 1}/{episodes}:\n")
        # Atstatome žaidimo lentą į pradinę būseną ir konvertuojame būseną į float32
        state = env.reset().astype(np.float32)
        done = False
        # Pradedame naują epizodą metrikų sekime
        metrics.new_episode()
        while not done:
            # Gaukite veiksmą iš agento
            action = agent.get_action(state)
            # Atlikite veiksmą ir gaukite kitą būseną, atlygį ir žaidimo pabaigos flagą
            next_state, reward, done = env.step(action)
            next_state = next_state.astype(np.float32)
            if done and reward == 0:
                reward = -1  # Neigiamas atlygis už pralaimėjimą
            # Atnaujiname metrikas
            metrics.update(reward, done, env.current_player)
            # Treniruojame agentą
            agent.train(state, action, reward, next_state, done)
            # Atnaujiname būseną
            state = next_state

        # Spaudiname būseną, veiksmą, atlygį, kitą būseną ir žaidimo pabaigos flagą
        print(f"Game:\n{next_state}\nReward: {reward}\n")
```

3 pav. *Žaidimo prieš save funkcijos realizacija*

3.3.5. Modelio vertinimas

Po treniravimo modelis buvo vertinamas žaidžiant prieš atsitiktinį agentą. Vertinimo metu buvo stebimos pergalės, pralaimėjimai ir lygiosios, siekiant nustatyti modelio efektyvumą.

```python
# Vertinimas prieš atsitiktinį agentą
def evaluate(agent, env, episodes):
    metrics = {"win": 0, "loss": 0, "draw": 0}
    for _ in range(episodes):
        # Atstatome žaidimo lentą į pradinę būseną ir konvertuojame būseną į float32
        state = env.reset().astype(np.float32)
        done = False
        while not done:
            if env.current_player == 1:
                # AI gauna veiksmą
                action = agent.get_action(state)
            else:
                # Atsitiktinis veikėjas gauna veiksmą
                action = random.choice(env.get_available_actions())
            # Atlikite veiksmą ir gaukite kitą būseną, atlygį ir žaidimo pabaigos flagą
            next_state, reward, done = env.step(action)
            next_state = next_state.astype(np.float32)
            if done:
                if reward == 1:
                    if env.current_player == 2:  # AI laimi
                        metrics["win"] += 1
                    else:  # AI pralaimi
                        metrics["loss"] += 1
                elif reward == 0.5:
                    metrics["draw"] += 1
            # Atnaujiname būseną
            state = next_state
    total = sum(metrics.values())
    win_rate = metrics["win"] / total
    loss_rate = metrics["loss"] / total
    draw_rate = metrics["draw"] / total
    print(
        f"Win Rate: {win_rate:.2f}, Loss Rate: {loss_rate:.2f}, Draw Rate: {draw_rate:.2f}"
    )
```

4 pav. *Modelio vertinimo funkcijos realizacija*

4 pav. esanti funkcija įvertino rezultatus, kad modelis galėtų išmokti optimalias žaidimo strategijas ir efektyviai žaisti kryžiukų-nuliukų žaidimą.

4. Gauti rezultatai

Rasti geriausią modelio variantą, buvo atlikti modelio modelių apmokymai, kiekvienam modeliui skiriant sužaisti 500 žaidimų prieš save. Kadangi mokymas išnaudoja daug laiko, buvo pasirinktas nedidelis sužaisti skirtų žaidimų skaičius. Gauti rezultatus buvo skirta jau anksčiau aprašyta ir **4 pav.** pavaizduota funkcija, kuri įvertina santykį tarp laimėtų, pralaimėtų ir baigtų lygiosiomis žaidimų. Naudojant skatinamąjį mokymąsi sunku yra įvertinti jo tikslumą ir reikšmę, todėl buvo pasirinkta vertinti būtent šiuos santykius.

1 lentelė. *Mokymo metu sužaistų žaidimų santykis*

Laimėjimai:	55%
Pralaimėjimai:	45%
Lygiosios:	0%

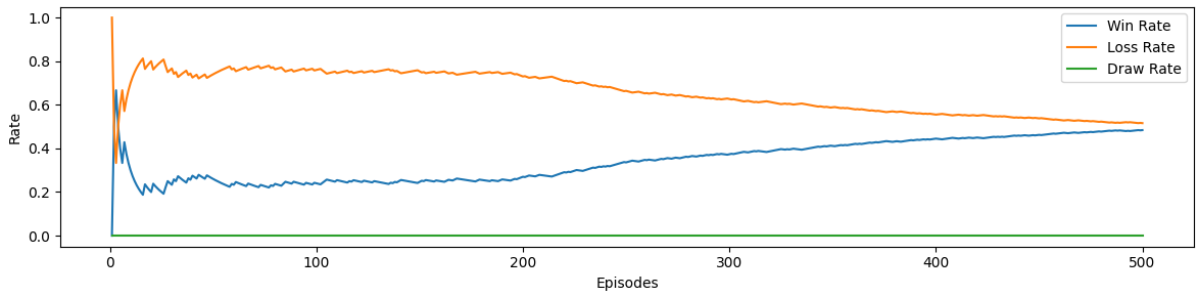
1 lentelė gauti rezultatai parodo, kad po 500 sužaistų su savimi žaidimų geriausias rastas modelis jau pasiekė 55% laimėjimų santykį. Tai rodo, kad toliau mokinant modelį galima pasiekti žymiai geresnius rezultatus. Tačiau vien šios lentelės neužtenka, kad galima būtų lyginti, kiek vienas modelis geresnis už kitą, nes galiausiai jie visi gali pasiekti tą patį laimėjimų santykį sužaidžiant panašų žaidimų skaičių.

Mokymosi greičio palyginimui ir santykių nuo žaidimų skaičiaus stebėjimui buvo sukurta papildoma klasė renkanti mokymo metrikas (**5 pav.**).

```python
# Metrikų sekimo klasė
class Metrics:
    def __init__(self):
        # Inicijuojame tuščius sąrašus epizodų atlygiams, laimėjimų, pralaimėjimų ir lygiųjų rodikliams
        self.episode_rewards = []
        self.win_rates = []
        self.loss_rates = []
        self.draw_rates = []

    def update(self, reward, done, current_player):
        # Atnaujiname einamojo epizodo atlygį
        self.episode_rewards[-1] += reward
        if done:
            # Jei žaidimas baigtas, atnaujiname atitinkamą rodiklį
            if reward == 1:
                if current_player == 1:  # AI laimi
                    self.win_rates[-1] += 1
                else:  # AI pralaimi
                    self.loss_rates[-1] += 1
            elif reward == 1:
                self.draw_rates[-1] += 1

    def new_episode(self):
        # Pridedame naujus įrašus kiekvienam metrikų sąrašui naujam epizodui
        self.episode_rewards.append(0)
        self.win_rates.append(0)
        self.loss_rates.append(0)
        self.draw_rates.append(0)

    def plot(self):
        # Parodome metrikų grafiką
        episodes = range(1, len(self.episode_rewards) + 1)
        plt.figure(figsize=(12, 6))
        # Pirmas grafikas rodo epizodinius atlygį
        plt.subplot(2, 1, 1)
        plt.plot(episodes, self.episode_rewards, label='Episode Reward')
        plt.xlabel('Episodes')
        plt.ylabel('Cumulative Reward')
        plt.legend()
        # Antras grafikas rodo laimėjimų, pralaimėjimų ir lygiųjų rodiklius
```

5 pav. *Metrikų stebėjimo klasės realizacija*

Naudojant šią klasę yra sugeneruojamas grafikas, kuris būtent ir parodo rezultatus.



6 pav. *Santykių priklausomybė nuo sužaistų žaidimų*

Pagal **6 pav.** ir **1 lentelė** gautus rezultatus galima pastebėti, kad nei vienas iš sužaistų žaidimų nebuvo baigtas lygiosiomis. Šis rezultatas gavosi, būtent tik mokinant šį geriausią

gautą modelį, bet ne su kitais modeliais. Taip pat galima pastebėti, kad nors mokymosi progresas yra ne toks greitas, modelis vis tiek toliau mokosi ir daro vis mažiau klaidų.

5. Išvados

Šiame projekte buvo sėkmingai sukurta kryžiukų-nuliukų žaidimo programa, naudojant daugiasluoksnį tiesioginio sklidimo neuroninį tinklą. Remiantis gautais rezultatais, galima padaryti keletą svarbių išvadų:

1. **Neuroninio tinklo veikimas:** Sukurtas neuroninio tinklo modelis, susidedantis iš trijų sluoksnių, parodė gerus rezultatus žaidžiant kryžiukų-nuliukų žaidimą. Naudojant paprastą architektūrą ir aktyvacijos funkcijas, modelis sugebėjo mokytis ir priimti teisingus sprendimus žaidimo metu.
2. **Apmokymo strategija:** Naudota savarankiško mokymosi strategija su Q-learning algoritmu leido modeliams mokytis žaisdami prieš save. Tai efektyvus būdas generuoti mokymo duomenis ir užtikrinti, kad modelis nuolat tobulėtų.
3. **Mokymosi progresas:** Nepaisant to, kad mokymosi progresas nebuvo greitas, modelis parodė gebėjimą mokytis ir daryti vis mažiau klaidų ilgalaikėje perspektyvoje. Tai rodo, kad toliau mokinant modelį galima pasiekti dar geresnių rezultatų, tačiau tam reikia daugiau laiko.

Nors gauti rezultatai yra perspektyvūs, yra galimybių toliau tobulinti modelį. Galima eksperimentuoti su sudėtingesnėmis architektūromis, kitomis mokymosi strategijomis ar hiperparametrų optimizavimu, siekiant dar didesnio modelio efektyvumo. Apskritai, šis projektas parodė, kad naudojant daugiasluoksnius neuroninius tinklus ir Q-learning algoritmą, galima efektyviai ir nesunkiai išmokyti modelius žaisti kryžiukų-nuliukų žaidimą.