Model sieci neurnonowej grającej w kości

Artur Dwornik, Szymon Wojturski, Maciej Malinowski

AGH University
June 16, 2024

1 Cel ćwiczenia

Celem ćwiczenia jest stworzenie modelu sieci neuronowej oraz wytrenowanie go tak, aby potrafił grać w kości.

2 Gra w kości

Gra w kości, znana również jako Liar's Dice, jest popularną grą towarzyską, w której uczestnicy starają się blefować i oceniać prawdopodobieństwo, aby wygrać. W grze bierze udział co najmniej dwóch graczy, z których każdy rzuca zestawem kostek i ukrywa wyniki przed innymi graczami.

2.1 Zasady gry

- Każdy gracz ma zestaw pięciu sześciennych kostek.
- Gracze rzucają swoimi kostkami i ukrywają wyniki przed przeciwnikami.
- Gra toczy się w turach, podczas których gracze składają deklaracje dotyczące liczby i wartości kostek znajdujących się pod kubkami wszystkich graczy.
- Deklaracje muszą stopniowo wzrastać. Na przykład, po deklaracji "trzy trójki" kolejny gracz może zadeklarować co najmniej "trzy czwórki" lub "cztery jedynki" (gdy wzrasta liczba kostek, można zadeklarować mniejszą liczbę oczek).
- W grze znajduje się "as", jest nim ścianka z jednym oczkiem. Może on zastąpić jakąkolwiek sciankę.

- Gracz może też zakwestionować poprzednią deklarację, twierdząc, że jest ona nieprawdziwa.
- Jeśli deklaracja jest prawdziwa, gracz, który ją zakwestionował, przegrywa rundę. Jeśli jest fałszywa, przegrywa gracz, który złożył fałszywą deklarację.

2.2 Gra z niepełną informacją

Gra w kości jest klasycznym przykładem gry z niepełną informacją. W takich grach gracze nie mają pełnej wiedzy o wszystkich elementach gry, co wpływa na podejmowane przez nich decyzje. W przypadku Liar's Dice, każdy gracz zna jedynie wyniki swoich własnych rzutów kostkami, ale nie zna wyników rzutów innych graczy. Ta niepełna informacja zmusza graczy do spekulacji i strategii opartych na przewidywaniach dotyczących rzutów przeciwników oraz ich zachowań.

2.3 Znaczenie gry z niepełną informacją

Gry z niepełną informacją, takie jak Liar's Dice, są szczególnie interesujące z punktu widzenia teorii gier i sztucznej inteligencji, ponieważ wymagają od graczy (lub algorytmów) radzenia sobie z niepewnością i podejmowania decyzji na podstawie niepełnych danych. W takich grach kluczowe jest:

- **Przewidywanie zachowań przeciwników:** Gracze muszą oceniać, czy deklaracje innych graczy są prawdziwe, czy fałszywe, na podstawie ich wcześniejszych zachowań i tendencji do blefowania.
- **Strategiczne blefowanie:** Gracze muszą umiejętnie wprowadzać przeciwników w błąd, składając fałszywe deklaracje, które są trudne do zakwestionowania.
- **Zarządzanie ryzykiem:** Każda decyzja wiąże się z ryzykiem, a gracze muszą balansować między składaniem wyższych deklaracji a ryzykiem zostania zakwestionowanym.

3 Dotychczasowe rozwiązania

Nasz model głownie bazowaliśmy na modelu z Thomasa Dybdahl Ahle. Zastosował o reinforcement learning. Jest to metoda uczenia sieci neuronowej polegająca na zaimplementowaniu funkcji kary, którą model stara się minimalizować i tym samym ucząc się.

4 Opis danych

Z racji, że w naszym rozwiązaniu użyliśmy metody reinforcement learningu, model nie potrzebuje żadnych danych do nauki. Zastosowaną przez nas funkcją kary jest "Conterfactual Regret Mimization".

4.1 Conterfatual Regret Mimization

Counterfactual Regret Minimization (CFR) to iteracyjna metoda wykorzystywana w teorii gier, szczególnie w kontekście gier o niepełnej informacji, takich jak poker. CFR opiera się na pojęciu *żałowania* (regret), które mierzy, jak bardzo gracz żałuje, że nie podjął alternatywnej akcji w danej sytuacji.

5 Reinforcement Learning

Reinforcement Learning (RL) to obszar uczenia maszynowego, w którym agenci uczą się podejmować decyzje poprzez interakcję ze środowiskiem w celu maksymalizacji długoterminowej nagrody (lub też minimalizowania kary).

5.1 Podstawowe pojęcia

- **Agent**: System podejmujący decyzje.
- **Środowisko**: Wszystko, z czym agent wchodzi w interakcje.
- **Stan (State)**: Aktualna sytuacja agenta w środowisku.
- **Akcja (Action)**: Działanie, które agent może podjąć.
- **Nagroda/Kara (Reward)**: Informacja zwrotna od środowiska na temat skuteczności podjętej akcji.
- **Polityka (Policy)**: Strategia, według której agent podejmuje decyzje.
- **Funkcja wartości (Value Function)**: Oczekiwana suma nagród, które można uzyskać z danego stanu.

5.2 Proces uczenia

RL polega na ciągłym dostosowywaniu polityki agenta na podstawie uzyskanych nagród. Proces ten można opisać następującymi krokami:

1. Agent obserwuje aktualny stan środowiska.
2. Agent podejmuje akcję zgodnie z obecną polityką.
3. Środowisko przechodzi do nowego stanu i dostarcza agentowi nagrodę.
4. Agent aktualizuje swoją politykę na podstawie uzyskanej nagrody i nowego stanu.

5.3 Algorytmy Reinforcement Learning

Istnieje wiele algorytmów RL, w tym:

- **Q-learning**: Algorytm off-policy, który uczy się funkcji wartości akcji.
- **SARSA (State-Action-Reward-State-Action)**: Algorytm on-policy, który również uczy się funkcji wartości akcji, ale bazuje na aktualnej polityce.
- **Deep Q-Networks (DQN)**: Zastosowanie sieci neuronowych do aproksymacji funkcji wartości w Q-learning.
- **Policy Gradient Methods**: Algorytmy, które bezpośrednio optymalizują politykę, zamiast funkcji wartości.

6 Architektura sieci

Sieć neuronowa zastosowana w tym ćwiczeniu to model `NetConcat`, który składa się z pięciu ukrytych w pełni połączonych warstw o liczbie neuronów odpowiednio: 500, 400, 300, 200, 100. Warstwy te wykorzystują funkcję aktywacji `relu`. Warstwa wyjściowa posiada jeden neuron z funkcją aktywacji `tanh`.

Layer (type)	Output Shape	Param #
dense (Dense)	?	78,500
dense_1 (Dense)	?	200,400
dense_2 (Dense)	?	120,300
dense_3 (Dense)	?	60,200
dense_4 (Dense)	?	20,100
dense_5 (Dense)	?	101

Total params: 479,601 (1.83 MB)
Trainable params: 479,601 (1.83 MB)
Non-trainable params: 0 (0.00 B)

Podsumowanie modelu

7 Trenowanie

7.1 Niestandardowa funkcja kroku trenowania

Gdy implementuje się model kategorii uczenia maszynowego reinforcement learning wymagane jest napisanie niestandardowej funkcji kroku trenowania. Na szczęście API TensorFlow jest w pełni wyposarzone i służy pomocą w tym zadaniu.

Oto implementacja pętli trenowania:

```
@tf.function
def train_step(privs, states, targets):
    with tf.GradientTape() as tape:
        predictions = model(privs, states, training=True)
        loss = loss_fn(targets, predictions)
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
    return loss
```

Działane funkcji krok po kroku:

1. **Dekorator** `@tf.function`: Informuje TensorFlow, że ta funkcja powinna być skompilowana jako graf obliczeniowy, co poprawia jej wydajność.

2. **Definicja funkcji** `train_step`: Funkcja `train_step` przyjmuje trzy argumenty: `privs` (prywatne dane wejściowe), `states` (stany gry) oraz `targets` (docelowe wartości).

3. **Kontekst** `tf.GradientTape`: Tworzy kontekst, w którym TensorFlow śledzi wszystkie operacje, aby później móc obliczyć gradienty.

4. **Obliczanie predykcji**: Wywołanie modelu z `privs` i `states` w trybie treningowym (`training=True`). Model zwraca przewidywane wartości (`predictions`).

5. **Obliczanie straty**: Użycie funkcji straty (`loss_fn`), która oblicza błąd między wartościami docelowymi (`targets`) a przewidywanymi (`predictions`).

6. **Obliczanie gradientów**: Funkcja `tape.gradient` oblicza gradient straty w odniesieniu do trenowalnych zmiennych modelu (`model.trainable_variables`).

7. **Aktualizacja wag modelu**: Optymalizator (`optimizer`) aktualizuje wagi modelu na podstawie obliczonych gradientów, wykorzystując metodę `apply_gradients`.

8. **Zwracanie straty**: Funkcja zwraca wartość straty (`loss`).

7.2 Proces trenowania

Model jest trenowany za pomocą optymalizatora Adam z szybkością uczenia 10^{-3}. Funkcją straty jest średni błąd kwadratowy (Mean Squared Error).

Implementacja głównej pętli trenowania:

```
def train():
    all_rolls = list(itertools.product(game.rolls(0), game.rolls(1)))
    for t in range(1000):
        replay_buffer = []

        BS = 100  # Number of rolls to include
        for i, (r1, r2) in enumerate(
            all_rolls if len(all_rolls) <= BS else random.sample(all_rolls, BS)
        ):
            play(r1, r2, replay_buffer)

        random.shuffle(replay_buffer)
```

```
privs, states, y = zip(*replay_buffer)

privs = tf.stack(privs)
states = tf.stack(states)
y = tf.constant(y, dtype=tf.float32)
y = tf.reshape(y, (-1, 1))

loss = train_step(privs, states, y)
with open("losses_test.txt", "a") as f:
    f.write(f"{t} {loss.numpy()}\n")

# Compute and print loss
print(t, loss.numpy())

model.save_weights("model.h5")
```

Funkcja `train` jest główną pętlą treningową dla modelu sieci neuronowej stosowanego do gry w Liar's Dice. Poniżej znajduje się krok po kroku opis działania tej funkcji:

1. **Generowanie wszystkich możliwych rzutów kostką:**

   ```
   all_rolls = list(itertools.product(game.rolls(0), game.rolls(1)))
   ```

   Funkcja generuje listę wszystkich możliwych kombinacji rzutów kostką dla dwóch graczy.

2. **Pętla treningowa:**

   ```
   for t in range(1000):
   ```

   Rozpoczyna się główna pętla treningowa, która będzie wykonywana przez 1000 iteracji.

3. **Inicjalizacja bufora powtórek:**

   ```
   replay_buffer = []
   ```

   Bufor powtórek (*replay buffer*) jest inicjalizowany jako pustą listę.

4. **Określenie liczby rzutów do uwzględnienia:**

   ```
   BS = 100  # Number of rolls to include
   ```

   Liczba rzutów, które będą uwzględnione w każdej iteracji, jest ustawiona na 100.

5. **Iteracja przez kombinacje rzutów:**

```
for i, (r1, r2) in enumerate(
    all_rolls if len(all_rolls) <= BS else random.sample(all_rolls, BS)
):
    play(r1, r2, replay_buffer)
```

Jeśli liczba możliwych kombinacji rzutów jest mniejsza lub równa 100, iteracja przechodzi przez wszystkie kombinacje. W przeciwnym razie, wybiera losowo 100 kombinacji. Każda kombinacja rzutów jest następnie używana do rozegrania jednej gry, a wyniki są dodawane do bufora powtórek.

6. **Losowe przetasowanie bufora powtórek:**

```
random.shuffle(replay_buffer)
```

Bufor powtórek jest losowo przetasowany, aby zapobiec kolejności, która mogłaby negatywnie wpłynąć na proces uczenia.

7. **Rozpakowanie danych z bufora powtórek:**

```
privs, states, y = zip(*replay_buffer)
```

Dane z bufora powtórek są rozpakowane do trzech zmiennych: `privs` (prywatne informacje), `states` (stany gry) oraz `y` (docelowe wartości).

8. **Przygotowanie danych wejściowych do modelu:**

```
privs = tf.stack(privs)
states = tf.stack(states)
y = tf.constant(y, dtype=tf.float32)
y = tf.reshape(y, (-1, 1))
```

Prywatne informacje i stany gry są przekształcone w tensory za pomocą funkcji `tf.stack`. Docelowe wartości są przekształcone w tensor typu `float32` i zreshape'owane do odpowiedniego kształtu.

9. **Trening modelu:**

```
loss = train_step(privs, states, y)
```

Funkcja `train_step` jest wywoływana, aby przeprowadzić jeden krok treningowy modelu, a wartość straty jest zapisywana.

10. **Zapis wartości straty do pliku:**

```
with open("losses_test.txt", "a") as f:
    f.write(f"{t} {loss.numpy()}\n")
```

Wartość straty dla bieżącej iteracji jest zapisywana do pliku tekstowego `losses_test.txt`.

11. **Wyświetlanie wartości straty:**

```
print(t, loss.numpy())
```

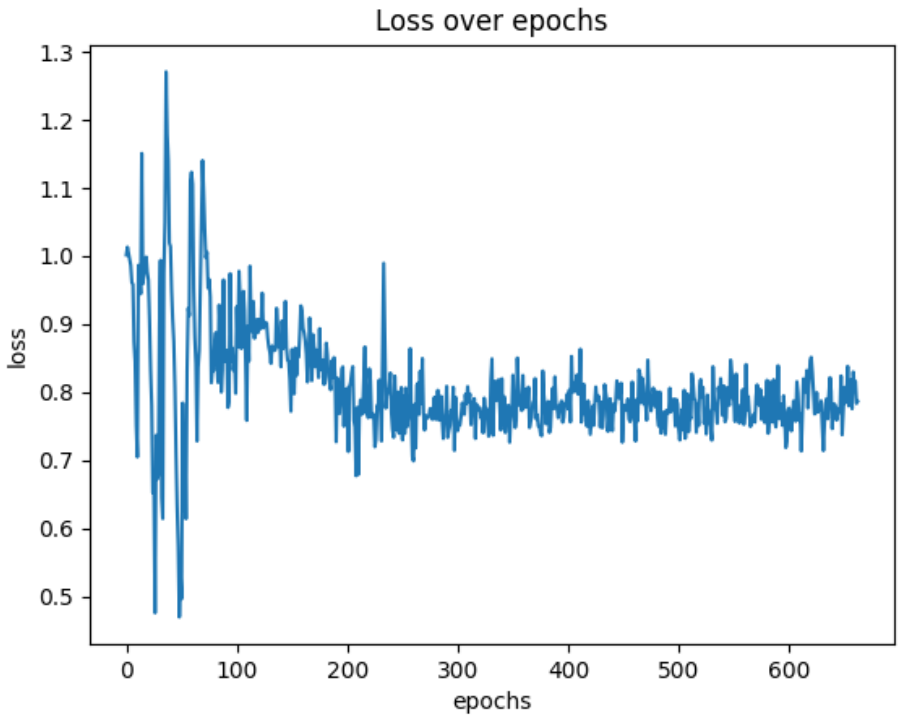
Wartość straty dla bieżącej iteracji jest wyświetlana w konsoli.

12. **Zapis wag modelu:**

```
model.save_weights("model.h5")
```

Wagi modelu są zapisywane do pliku `model.h5`.

Wykres funkcji straty od ilości epok:



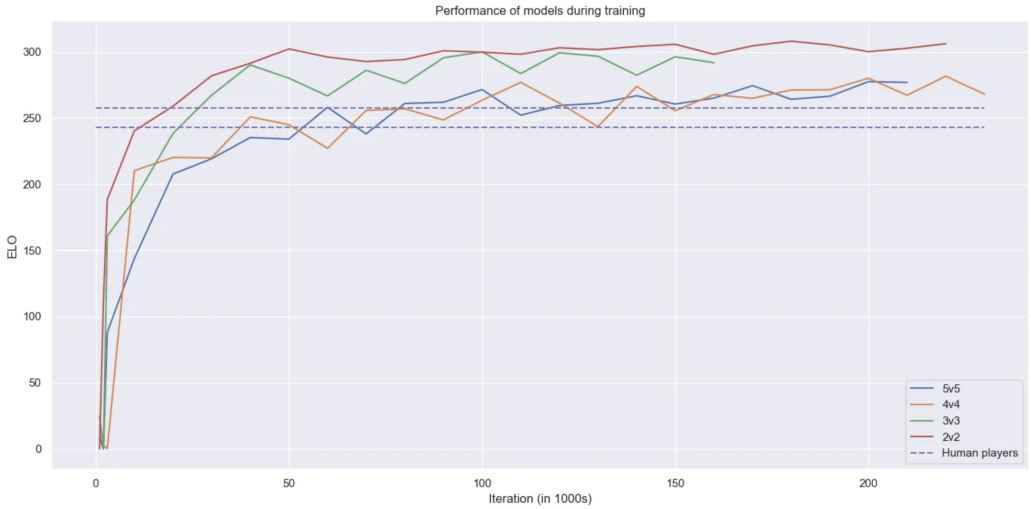
Wykres funkcji straty od ilości epok

7.3 Wnioski z trenowania

Jak możemy zaobserwować z wykresu funkcji straty (loss) skacze. Jest to spowodowane dość dużyą szybkością uczenia (10^{-3}). Model możnaby usprawnić dodając warstwy "dropout" oraz stworzenie niestandardowego callbacku, który dynamicznie zmieniałby szybkość uczenia w zależności od różnicy wartości funkcji strady w sąsiadujących epokach. Możemy równie zaobserwować, że przy około setnej epoce model zaczyna robić znaczne postępy w umiejętności grania w kości. Warto również zaznaczyc, że w przeciwieństwie do modeli sieci neuronowych takich rozpoznających obrazy, my **nie możemy doprowadzić loss do wartości bliskich zera**. Jest to spowodowane tym, że w każdej iteracji model gra na bardzo delikatnie gorszą wejsę siebie, przez co ma szanse na wygraną bliskie 50%.

8 Zestawienie wyników z dotychczasowymi rozwiązaniami

Niestety, autor artykułu nie wystawił API, dzięki któremu moglibysmy porównać nasze modele. Jednak podejrzewamy, że nasz model nie odbiega znacząco od modelu autora, przez pewne podobieństwa w metodyce uczenia oraz architekturze modelów.



Wykres umiejętności modelu od ilosći iteracji (w tysiącach)

9 Dane techniczne sprzętu

Do treningu modelu wykorzystano następujący sprzęt:
MacBook Pro 2020 (Intel)

- Procesor: 1,4 GHz Quad-Core Intel Core i5
- Pamięć RAM: 8 GB 2133 MHz LPDDR3
- GPU: Intel Iris Plus Graphics 645 1536 MB
- System operacyjny: macOS Sonoma 14.5

10 Podział pracy

Podział pracy między członków zespołu wyglądał następująco:

- Implementacja modelu: Artur, Szymon, Maciej

- Zaimplementowanie funkcji kary: Artur
- Przeprowadzenie eksperymentów: Szymon
- Interfejs graficzny: Maciej
- Dokumentacja: Artur

11 Źródła

- *Liar's Dice by Self-Play* Thomas Dybdahl Ahle towardsdatascience.com
- *TensorFlow Guide* TensorFlow, Google
- *An Introduction to Counterfactual Regret Minimization* Todd W. Neller Marc Lanctot pdf