Lernen des Spiels TicTacToe durch ein neuronales Netz mit Deep Q-Learning

Jan Bessler
d.velop AG
jan.bessler04@gmail.com

Simon Krempin
d.velop AG
simon.krempin@gmail.com

I. Einführung

Diese Ausarbeitung wurde im Rahmen der Veranstaltung „Projekte und Fallstudien in der Wirtschaftsinformatik“ an der IHK erstellt. Ziel der Vorlesung ist eine Einführung in verschiedene Algorithmen zur Mustererkennung und wie man diese in einem praktischen Kontext anwenden kann.

II. Problemstellung

Die Aufgabe ist es ein auschließlich selbstlernendes Programm zu schreiben, welches, nach dem Lernprozess, in der Lage sein soll, Tic-Tac-Toe zu spielen. Dabei ist eine Anforderung, wenn möglich, den Agenten mit einem neuronalen Netz zu implementieren. Verboten ist die explizite Verwendung von festgeschriebenen Spielbäumen oder Strategien (wie dem Minmax- oder Alpha-Beta-Verfahren) im Agenten. Der Agent muss also auschließlich aus der Interaktion mit der Umgebung eine eigene Strategie entwickeln. Der Agent darf von der Umgebung erfahren welche Züge valide sind, und wann das Spiel zuende ist. Abschließend muss der Agent in der Lage sein, sowohl als X als auch als O spielen zu können.

III. Mögliche Spielsituationen

Als mögliche Spielsituation ist jeder Stand des Spielfeldes definiert, der durch ein legales und abwechselndes Spiel entstehen kann. Zum Zählen der Spielstände wird ein rekursiver Algorithmus genutzt, der jeden möglichen, validen und einzigartigen Spielstand in einem Baum generiert.

Algorithm 1: Count States

```
1: history ← []
2: procedure COUNT-STATES(state, turn)
3:     states ← 0
4:
5:     for i, v ∈ state do
6:         if v == −1 ‖ v == 1 then
7:             continue
```

```
        end
        new_state ← state.copy()
        new_state [i] ← turn
        new_turn ← −turn
        if new_state ∈ history then
            continue
        else
            history.append(new_state)
        end
        if is_over(new_state) then
            states ← states + 1
        else
            state ← 1 + Count-States(new_state,new_turn)
        end
    end
    return states
end
```

Wobei turn ein Integer ist, der die Werte -1, 1 und 0 annehmen kann (für die beiden Symbole und ein leeres Feld respektive), und state eine Liste an Integern ist, die das Spielfeld in flacher Form repräsentiert. Mit diesem Algorithmus werden 5477 valide Spielstände identifiziert, wobei noch einer addiert werden muss für ein vollkommen leeres Spielfeld. Somit ist die Anzahl möglicher Spielstände 5478. Für diese Berechnung wird die Annahme getroffen, dass es einen ersten und einen zweiten Spieler gibt, unabhängig von ihren Symbolen. Beide Spieler können als X oder O spielen, sodass es irrelevant ist welches der Symbole für den ersten Zug genutzt wird. Es wird also der erste Zug auf einem Feld nicht für X und O jeweils als separate Spielstände gezählt, sondern ein Mal für den ersten Spieler.

IV. Reinforcement Learning

Das bestärkende Lernen (eng. Reinforcement Learning) ist eine Art des maschinellen Lernens, die neben dem überwachten und dem unüberwachten Lernen zu sehen ist. Beim überwachten Lernen erhält ein Agent einen Trainingsdatensatz mit beschrifteten Trainingsdaten. Durch diesen lernt der Agent generelle Muster und kann so nach dem Training auch mit unbeschrifteten Daten umgehen. Beim unüberwachten Lernen identifiziert ein Agent Muster aus großen Mengen unbe-

schrifteter Daten. Im Gegensatz zu diesen beiden Lernmethoden ist es das Ziel des Agenten beim bestärkenden Lernen die Regeln oder die Taktik in einer Umgebung zu erkennen und auszunutzen. Der Agent startet mit gar keinem oder minimalem Wissen und erlernt durch Interaktion mit seiner Umgebung und Belohnungen, wie er seine erhaltene Belohnung maximieren kann. Dazu nimmt der Agent seine Umgebung wahr und führt auf Basis seiner Beobachtung eine Aktion aus. Das Resultat einer Aktion ist immer eine Belohnung. Je nach Höhe dieser Belohnung erkennt der Agent welche Aktionen vorteilhaft sind und welche nicht.

Kritisch beim bestärkenden Lernen ist das Verhältnis von Erkundung und Ausnutzung der Strategie. Der Agent maximiert bei der Ausnutzung seine Belohnung, indem er die Aktion ausführt, die ihm bekanntlich die höchste Belohnung bringt. Um jedoch seine Strategie potenziell zu verbessern, muss der Agent teilweise neue Aktionen abseits seiner bekannten Strategie nehmen. Das ist die Erkundung.

A. Q-Learning

Der grundsätzliche Zyklus beim bestärkenden Lernen ist die Wahrnehmung der Umgebung, das Ausführen einer Aktion, das Erhalten einer Belohnung und das Justieren der Strategie. Diese Schritte sind für jeden bestärkend lernenden Algorithmus gleich. Wie aber die Implementierung aussieht kann unterschiedlich sein. Eine Implementierung des bestärkenden Lernens ist das Q-Learning.

Beim Q-Learning wählt ein Agent seine nächste Aktion basierend auf dem Q-Wert (auch Qualität) einer Aktion in seinem aktuellen Zustand. Beim klassischen Q-Learning kennt der Agent den Wert einer Aktion durch eine Tabelle, in der jede Zustand-Aktion-Kombination gespeichert und gepflegt wird. Durch erhaltene Belohnungen und mit Hilfe der Bellman-Gleichung werden die Q-Werte in der Tabelle des Agenten mit der Zeit justiert. Schlussendlich sollten dem Agenten die gelernten Werte in der Q-Tabelle eine optimale Strategie ermöglichen.

Die Bellmann-Gleichung erfordert, dass die optimale zukünftige Belohnung für einen Zustand berechnet oder geschätzt werden kann. In Situationen mit wenigen Zuständen ist das einfach und sinnvoll durch eine Tabelle lösbar. Jedoch ist der Ansatz mit einer Q-Tabelle nicht sinnvoll, wenn es viele Zustände gibt (zum Beispiel in einer Umgebung mit kontinuierlichen Daten). In diesen Fällen kann die zukünftige Belohnung durch ein neuronales Netz geschätzt werden.

B. Deep Q-Learning

Beim Deep Q-Learning füttern die Sensoren des Agenten in ein neuronales Netz, das dann die Aktion des Agenten bestimmt. Dafür hat das Netz Eingabeneuronen, die den einzelnen Sensoren des Agenten entsprechen, und Ausgabeneuronen, die den möglichen Aktionen entsprechen. Durch die Belohnung, die der Agent nach jedem Schritt bekommt, werden die Gewichte im Netz regelmäßig mit Hilfe von Backpropagation justiert, um in Zukunft höhere Belohnungen zu erzielen.

V. Implementierung

In unserer Implementation haben wir uns für den Deep Q-Learning Ansatz entschieden, da die Aufgabenstellung fordert ein neuronales Netz zu nutzen sofern möglich. Dabei sei angemerkt, dass ein Q-Table-basierter Ansatz hier, aufgrund der relativ geringen Anzahl an Zuständen, vollkommen ausreichen würde.

Der Deep Q-Learning-Agent basiert auf einem neuronalen Netz mit neun Eingabeneuronen, zwei versteckten Schichten mit 64 und 32 Neuronen respektive und einer Ausgabeschicht mit neun Neuronen. Jede Schicht ist mit der ReLU-Funktion konfiguriert, bis auf die letzte, welche die Werte linear ausgiebt. Das ermöglicht es dem Netz die Q-Werte unverfälscht zu bestimmen. Jedes Neuron der Eingabeschicht entspricht einem Feld des TicTacToe Spielfeldes und erhält, je nach Zustand des Feldes, die Werte 1, −1 oder 0.[1] Welches der beiden Symbole für welchen Zahlenwert steht ist irrelevant für das Netz. Wichtig ist jedoch, dass der Agent mit beiden Symbolen spielen kann. Daher wird der Spielzustand entsprechend angepasst (Das Symbol des Agenten entspricht immer 1), bevor er dem Netz eingegeben wird.[2]

```
def get_action(self, state: list[int], valid_moves:
list[int], training=True) -> int:
  """Select action using epsilon-greedy policy"""
  if training and random() < self.epsilon:
    return np.random.choice(valid_moves)

  # Transform state to agent's perspective (agent=1,
opponent=-1)
```

[1] Eine Alternative hier wäre es drei Neuronen pro Feld zu konfigurieren, welche jeweils ein gegnerisches Feld, ein eigenes Feld und ein leeres Feld mit einer 1 repräsentieren. Das kann dabei helfen die Werte im Netz zu normalisieren.

[2] Eine Alternative wäre es ein weiteres Eingabeneuron dem Netz hinzuzufügen, welches für eine Partie das Symbol des Agenten repräsentiert.

```
  agent_state = [cell * self.player for cell in state]
  q_values =
self.model.predict(np.array([agent_state]), verbose=0)
[0]

  # Mask invalid moves
  masked_q = np.full(9, -np.inf)
  for move in valid_moves:
    masked_q[move] = q_values[move]
  return np.argmax(masked_q)
```

Um die Explorationsphase des Lernprozesses optimal zu nutzen, werden invalide Aktionen herausgefiltert. Das hilft dem Agenten eine Strategie zu lernen, anstatt nur zu lernen welche Aktionen valide sind und welche nicht. Zudem steht in der Aufgabenbeschreibung geschrieben, dass der Agent dieses Wissen bereits vor dem Lernprozess haben darf.

Wie schon in der Funktion erkennbar nutzt der Algorithmus eine Epsilon-Greedy Strategie. Der Wert von Epsilon bestimmt zu wie viel Prozent zufällige Aktionen gewählt werden müssen. Mit Voranschreiten des Lernprozesses wird Epsilon immer weiter vermindert, sodass der Agent immer mehr seine gelernte Strategie anwenden kann. Wobei aber Epsilon meist nicht bis auf Null reduziert wird. Mit welcher Geschwindigkeit Epsilon reduziert wird, was der Anfgangswert und der Minimalwert sind, sind Hyperparameter und können vor dem Training definiert werden. Wenn der Agent sich nicht mehr im Training befindet, wird Epsilon sinngemäß auf Null gesetzt, um ein optimales Spiel zu ermöglichen.

```
def train_agent(agent: TicTacToeAgent, opponent:
TicTacToeAgent, episodes=10000,
update_target_every=250):
  """Train the DQN agent"""
  agent = DQNAgent()

  for episode in range(episodes):
    agent.player = choice([X, O])
    opponent.player = -agent.player

    exp_x, exp_o, winner = play_game(agent, opponent,
training=True)
    exp = exp_x if agent.player == X else exp_o
```

```
    # Store experiences in replay memory for training
agent
    for e in exp:
      agent.remember(*e)

    # Train the agents network for one step
    agent.replay()

    # Update the target network less frequently to have
some
    # stability during training
    if episode % update_target_every == 0 and episode >
0:
      agent.update_target_model()
```

Das Training erfolgt, wie im obigen Codeausschnitt dargestellt, über mehrere Tausend Episoden (meist zwischen 2500 und 5000). Eine Episode umfasst einen Spieldurchlauf. Der Agent tritt in jeder Episode mit einem zufällig zugewiesenem Symbol gegen einen Gegner an. Um dem Agenten eine möglichst gute Taktik beizubringen, ist der Trainingspartner eine Implementation eines perfekten Spielers. In jeder Episode sammelt der Agent Erfahrungen, bestehend aus einem Zustand, der ausgeführten Aktion in diesem Zustand, der durch die Aktion erhaltenen Belohnung, dem Folgezustand und einem Indikator, ob das Spiel beendet wurde. Diese Erfahrungen werden gesammelt, um sie im nächsten Schritt zum Lernen zu verwenden. Der Agent besitzt einen rollierenden Erfahrungsspeicher mit einer maximalen Kapazität von 20000 Datensätzen. Nach jeder Episode wird das neuronale Netz basierend auf den gesammelten Erfahrungen justiert. Dazu wird mini-batch-Lernen verwendet.

```
def replay(self):
  """Train on batch from memory"""
  if len(self.memory) < self.batch_size:
    return

  minibatch = sample(self.memory, self.batch_size)

  states = np.array([exp[0] for exp in minibatch])
  actions = np.array([exp[1] for exp in minibatch])
  rewards = np.array([exp[2] for exp in minibatch])
```

```
  next_states = np.array([exp[3] for exp in minibatch])
  dones = np.array([exp[4] for exp in minibatch])

  # Get current Q values
  current_q = self.model.predict(states, verbose=0)

  # Get next Q values from target model
  next_q = self.target_model.predict(next_states,
verbose=0)

  # Update Q-value for the specific action taken
  for i in range(self.batch_size):
    if dones[i]:
      current_q[i][actions[i]] = rewards[i]
    else:
      current_q[i][actions[i]] = rewards[i] +
self.gamma * np.max(next_q[i])

  # Train model
  self.model.fit(states, current_q, epochs=1,
verbose=0)
```

Aus allen gesammelten Erfahrungen wird hierbei zufällig ein Batch zusammengestellt, aus dem das Netz lernen soll. Das hilft dabei die starke sequenzielle Korrelation zwischen den Erfahrungen aufzubrechen und somit das Lernen effizienter zu gestalten. Zudem hilft es dem „katastrophalen Vergessen“ entgegenzuwirken, bei dem der Agent bereits gelerntes wieder vergisst, weil es nicht mehr in seinen Trainingsdaten vorkommt. Ein weiterer Hyperparameter, Gamma, wird beim Lernen verwendet, um das Vorrausschauen des Agenten zu kontrollieren. Es bestimmt wie der Agent zukünftige Belohnungen im Vergleich zu sofortigen gewichtet.

Ein weiterer wichtiger Faktor im Lernprozess ist das Unterteilen zwischen dem aktuellen- und dem Zielnetzwerk. Das aktuelle Netz des Agenten wird nach jeder Episode durch Backpropagation angepasst. Das Zielnetzwerk dient dazu dem aktuellen Netzwerk ein Ziel zu geben, an das es sich anpassen kann. Das

[3]Eine modernere Alternative ist es ein Soft-Update des Zielnetzwerkes zu machen. Es also auch jede Episode zu aktualisieren, dafür aber jedes Mal nur ein kleines bisschen.

Zielnetzwerk wird daher seltener aktualisiert (alle paar Hundert Episoden).[3] So wird der Lernprozess stabil gehalten.

```
def train_agent(agent: TicTacToeAgent, opponent:
TicTacToeAgent, episodes=10000,
update_target_every=250):
  """Train the DQN agent"""
  agent = DQNAgent()

  for episode in range(episodes):
    ...

    agent.replay()

    if episode % update_target_every == 0 and episode >
0:
      agent.update_target_model()
```

VI. Probleme und Lösungen

A. *Katastrophales Vergessen*

Regelmäßig konnten wir beobachten, dass der Agent eine gute Strategie findet und nahezu mit dem perfekten Agenten gleichzieht, nur um ein paar Tausend Episoden später wieder auf sein Anfangsniveau zurückzukehren. Dieses Problem haben wir behoben, indem wir die Kapazität des Erfahrungsspeichers verdoppelt haben. Wir vermuten, dass wichtige Phasen aus dem Speicher „geschoben“ wurden, und somit die Strategie des Agenten beeinträchtigt wurde.

B. *Instabilität beim Lernen*

Der Agent hatte anfangs oft Phasen, in denen seine Performance stark geschwankt hat. Teilweise hat die Leistung auch stagniert. Um dem entgegenzuwirken haben wir die Lernrate auf 0.001 und den Gamma-Verfall auf 0.003 reduziert. Das hat für einen stabileren und ausführlicheren Lernprozess gesorgt. Die Parameter des neuronalen Netzes haben sich wohl zu schnell und zu viel verändert. Gleichzeitig war die Explorationsphase zu kurz, sodass der Agent gezwungen war, sich auf eine suboptimale Lösung festzulegen.

VII. Abschluss

In Zuge der Ausarbeitung konnte erfolgreich ein Deep Q-Learning Agent für das Spiel TicTacToe erstellt werden. Mithilfe einer Lernumgebung und einem perfekten Spieler als Trainer hat das neuronale Netzwerk die Regeln des Spiels erlernt und ist mit dem perfekten Spieler in der Performance gleichgezogen. Dabei haben klassische Probleme des Deep Q-Learning den Agenten herausgefordert, wie katastrophales Vergessen und Lern-Instabilität. Die Herausforderungen haben uns einen ersten, praktischen Einblick in die komplexe Domäne des Parameter-Fine-Tunings gegeben. Diese Erkenntnisse und Erfahrungen können wir nachhaltig in Folgeprojekte, oder im generellen Arbeitsalltag, einbringen.

Daneben konnten wir durch die Implementation unsere Kenntnisse in der Programmiersprache Python vertiefen, und unser Standbein in einer der Grundtechnologien von künstlicher Intelligenz festigen. Auch diese Komponente ist ein wichtiger Mehrwert für den anstehenden Berufsalltag.

References

[1] R. S. Sutton, A. G. Barto, and others, *Reinforcement learning: An introduction*, vol. 1, no. 1. MIT press Cambridge, 1998.

[2] R. Jagtap, "Understanding the Markov Decision Process (MDP)," 2024, [Online]. Available: https://builtin.com/machine-learning/markov-decision-process

[3] C. J. C. H. Watkins, "Learning from Delayed Rewards," 1989.

[4] A. Singh, "Reinforcement Learning: Bellman Equation and Optimality (Part 2)," 2019, [Online]. Available: https://medium.com/data-science/reinforcement-learning-markov-decision-process-part-2-96837c936ec3

[5] V. Mnih *et al.*, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.