Spiele mit Greenfoot und Reinforcement Learning

Inhaltsverzeichnis

Ausprobieren (1)

1. Lege per Mausklick ein Objekt der Klasse BreakoutGame an. Du kannst dann ein einfaches Spiel spielen, gesteuert mit den Tasten A und D.

2. Lege per Mausklick ein Objekt der Klasse AutoGame an. Du kannst dann ein einfaches Spiel spielen, gesteuert mit den Tasten A und D. Das Spiel ist nicht sehr interessant, und man muss sich die Geschwindigkeit herabsetzen.

3. Lege per Mausklick ein Objekt der Klasse Snake Game an. Du kannst dann ein einfaches Spiel spielen, gesteuert mit den Tasten ASDW.

Ausprobieren (2)

Lege per Mausklick ein Objekt der Klassen BreakoutGameKI, AutoGameKI oder SnakeGameKI an. Dann läuft ein kurzes Beispiel-Szenario mit einer lernenden KI ab. Es gibt dabei sehr viele Konfigurationsmöglichkeiten; die Unterklasse AutoScenarios, BreakoutScenarios und SnakeScenarios bieten einige vorkonfigurierte Szenarien an.

Allgemeines Vorgehen

Schritt 1:

Man erstellt ein einfaches grafisches Spiel mit Greenfoot. Beispiele: AutoGame, BreakoutGame, SnakeGame. Das Spiel sollte, weil es das einfacher macht, fast alles Nötige im Konstruktor erstellen, nur das Erstellen des zu steuernden Objekts, also Auto oder Schläger – sollte in einer setup-Methode ausgelagert sein, die vom Konstruktor aufgerufen wird. Dann können später Unterklassen leicht den Großteil der Initialisierungsarbeit der Oberklasse überlassen und nur die setup-Methode überschreiben.

Es ist außerdem günstig, aber ebenfalls nicht nötig, alle zu dem Spiel gehörenden Actor-Unterklassen in einer gemeinsamen Oberklasse zu sammeln, siehe die Klassen AutoElement, BreakoutElement oder SnakeElement.

Schritt 2:

Wenn das Spiel steht, macht man die Spielwelt, die bisher eine Unterklasse von World war, zu einer Unterklasse von AbstractGameWorld. Das Spiel sollte sich weiterhin wie gewohnt spielen lassen, wenn man Greenfoot die eigentliche Spielklasse als World verwenden lässt.

Schritt 3:

Man erstellt eine Unterklasse zur Spieleklasse, zum Beispiel AutoKI, BreakoutKI oder SnakeGameKI. Diese Klassen müssen, um sinnvoll arbeiten zu können, bestimmte ererbte Methoden überschreiben. Das ist die Hauptaufgabe, die im nächsten Abschnitt erklärt wird.

Schritt 4:

Da es so viele Varianten gibt, die man ausprobieren möchte, legt man sich eine Unterklasse der KI-Klasse an (die ja selber eine Unterklasse von AbstractGameWorld ist), um so einigermaßen elegant die Varianten anbieten zu können.

Die KI-Klasse und ihre Methoden

Beispiel: AutoGameKI, BreakoutGameKI, SnakeGameKI und BreakoutGameKIMinimal. Das letztere ist eine abgespeckte Klasse zur Veranschaulichung, die wir jetzt der Reihe nach durchgehen. Der Code passt auf eine Textseite.

Die ganz einfachen Methoden

public void setup()

- Überschreibt lediglich die Methode Oberklasse, die als Teil des Konstruktors dort aufgerufen wird.
- Ein Objekt vom Typ SchlaegerKI wird platziert, der sich vom ursprünglichen Schläger nur dadurch unterscheidet, dass seine act-Methode (mit der er sonst vom Spielenden gesteuert würde) leer ist.

- Die Greenfoot-World-Methode setActOrder wird aufgerufen, damit sichergestellt ist, dass Objekte der Klasse AnzeigeNextMoves nach allen anderen drankommen. Das muss man nicht machen, aber nur dann stimmt die Angabe des kommenden Spielzugs.
- Der Aufruf von setPlayers(new Agent []{ new Agent(0.0) }) legt fest, dass ein Standardagent den Spielpart übernehmen soll. Standard heißt: Reinforcement Learning mit einem Q-Table. Man kann auch mehrere Agenten für mehrere zu bewegende Objekte haben. Der NeuralAgent benutzt ein Q-Netz für das Reinforcement Learning.

public void erhoeheLevel()

- Ist nur dazu dazu, um die ererbte Methode, mit der man im regulären Spiel zum nächsten Schwierigkeitsgrad kommt, zu überschreiben.

private double berechneEntfernung(int x, int y, int x2, int y2)

- Eine Hilfsmethode, die die Entfernung zweier Punkte in einem kartesischen Koordinatensystem berechnet.

Die einfachen überschreibenden KI-Methoden

Diese Methoden werden von AbstractGameWorld und verschiedenen weiteren Klassen verwendet.

- public int [] getLegalMoves() { return new int [] { 1, 2, 0}; }
  Die verschiedenen Spielzüge oder Entscheidungsmöglichkeiten, zwischen denen die KI sich entscheichen kann, werden als int modelliert. Diese Methode gibt einfach ein Array zurück mit allen grundsätzlich möglichen Spielzügen.
- public String getNameForMove(int move)
  Zur Anzeige kann man sich statt der Zahlen für die Züge auch einen schöneren Namen geben lassen. Wird diese Methode nicht überschrieben, werden stattdessen einfach die Zahlenwerte dargestellt. Im Moment wird 0 auf "N" (nichts), 1 auf "L" (nach links gehen), 2 auf "R" (nach rechts gehen) abgebildet. Normalerweise würde man das mit einem Array-Attribut umsetzen.
- public void makeMove(int player, int move)
  Der eigentliche Zug: Was bedeuten die oben angelegten Zahlen in der Spiewelt? 1 bedeutet, dass der Schläger etwas nach links bewegt wird, 2 bedeutet, dass er etwas nach rechts bewegt wird. 0 bedeutet, dass er da bleibt, wo er ist. Andere Zahlen dürften nicht auftauchen, sie kommen ja nur aus dem Pool der legalen Züge.
  Bei einem Einspieler-Spiel wie im Beispiel wird der player-Parameter ignoriert.
- public double getInitialValue()
  Kann man auch weglassen. Hier geht es um den Startwert für die Inhalte der Q-Tabelle. Standard ist 1.

Die schwierigeren überschreibenden KI-Methoden

- public String getState() { return getState(0); }
  Hier geht es um den Zustand des Spiels. Manchmal ist der für jeden Spieler anders (bei unterschiedlichen Positionen einer Spielfigur, oder bei teilweise versteckter Information), manchmal ist er für alle identisch (Schach, Tic-Tac-Toe). Da es hier nur einen Zustand gibt statt unterschiedlicher Perspektiven, wird einfach die Perspektive von Player 0 aufgerufen.

- public String getState(int playerID)
  Eine der schwierigsten Entscheidungen: Wie man den Zustand codiert. Er solle zur leichteren Weiterverarbeitung später nur aus durch Doppelpunkte getrennten Ganzzahlen bestehen. Im Beispiel ist der Zustand einfach die Differenz zwischen x-Position von Kugel und Schläger. Damit ist viel wesentliche Information für eine Lösung enthalten, andere mögliche Zustände wären eine Kombinationen von x- und y-Position von Kugel und Schläger, und Blickrichtung der Kugel, eventuell noch ihre Geschwindigkeit, wenn es verschiedene Geschwindigkeiten gibt. Hier wird man viel experimentieren wollen. Deshalb enthält diese Methode in den tatsächlichen KI-Klassen auch meist einen switch, um einfach Verschiedenes ausprobieren zu können.

- public double getRewardWin() { return 10; }
  Bei der einfachsten Form des Lernen, in der Tabelle und ohne Neuronales Netz, wird in jedem Schritt überprüft, ob man gewonnen oder verloren hat. Nur wenn man gewonnen hat, gibt es eine positive Belohnung, deren Wert hier angegeben werden kann.

- public double getRewardLose() { return -5; }
  Wenn man verloren hat, gibt es eine negative Belohnung. Wenn das Spiel noch läuft, gibt es bei diesem einfachen Fall gar keine Belohnung.

- public int getWinner()
  Diese Methode wird regelmäßig aufgerufen und gibt zurück, ob man gewonnen hat (Rückgabewert 0, und das führt dann zur oben genannten positiven Belohnung), oder verloren hat (Rückgabewert 1, und das führt dann zur oben genannten negativen Belohnung), oder ob das Spiel noch läuft (Rückgabe -1, oder irgendetwas <0).
  Dieses einfache System eignet sich vor allem für Spiele mit ein oder zwei Agierenden. Bei dem Beispiel zählt als Gewinn, wenn der Schläger die Kugel berührt, und als Niederlage, wenn die Kugel im Aus ist. Alles andere ist Fortsetzung des Spiels. Belohnt wird hier also nicht kontinuierlich.

- public double getRewardForPlayer(int id)
  Hier wird statt de oben genannten Prinzips nicht nur bei Sieg oder Niederlage belohnt, sondern nach *jeder* einzelnen Entscheidung. Bei Agenten mit Neuronalen Netzen (NeuralAgent) muss das so sein, bei Q-Tabellen (Agent) kann man sich das aussuchen.
  Für das Breakout wird hier die 0 zurückgegeben, wenn sich die Kugel in der oberen Spielfeldhälfte befindet, 2 für die Berührung mit dem Schläger, und ansonsten eine negative Belohnung, deren Wert von der kartesischen Entfernung zwischen Kugel und Schläger abhängt. Auch hier kann man sich sehr viel verschiedene Varianten denken.

```
3   import greenfoot.*;
4
5   public class BreakoutGameKIMinimal extends BreakoutGame {
6
7       // Uberschreiben von Breakout-spezifische Methoden, und Hilfsmethoden
8
9       @Override
10      public void setup() {
11          setActOrder( new Class[]{ BreakoutElement.class, AnzeigeNextMoves.class });
12          schlaeger = new SchlaegerKI();
13          addObject(schlaeger, 360/2, 480-20);
14          setPlayers( new Agent []{ new Agent(0.0) } );
15      }
16
17      @Override
18      public void erhoeheLevel() {}
19
20      private double berechneEntfernung(int x, int y, int x2, int y2) {
21          return Math.sqrt((x-x2)*(x-x2)+(y-y2)*(y-y2));
22      }
23
24      // Uberschreiben von KI-relevanten Methoden
25
26      @Override
27      public int [] getLegalMoves() { return new int [] { 1, 2, 0}; }
28
29      @Override
30      public String getNameForMove(int move) {
31          if (move==0) return "N";
32          else if (move==1) return "L";
33          else if (move==2) return "R";
34          else return null;
35      }
36
37      @Override
38      public void makeMove(int player, int move) {
39          if      (move == 1) for (int i=0; i<5; i++) schlaeger.geheNachLinks();
40          else if (move == 2) for (int i=0; i<5; i++) schlaeger.geheNachRechts();
41          else { }
42      }
43
44      @Override
45      public double getInitialValue() { return 1; }
46
47      @Override
48      public String getState() {  return getState(0); }
49
50      @Override
51      public String getState(int irrelevantPlayerID) {
52          return "" + (schlaeger.getX() - kugel.getX())/10;
53      }
54
55      @Override
56      public double getRewardWin() { return 10; }
57
58      @Override
59      public double getRewardLose() { return -5; }
60
61      @Override
62      public int getWinner() {
63          if (kugel.beruehrtSchlaeger()) return 0;
64          else if (istImAus()) {
65              kugel.respawn();
66              return 1;
67          }
68          return -1;
69      }
70
71      @Override
72      public double getRewardForPlayer(int id) {
73        if (kugel.beruehrtSchlaeger()) return 2;
74        else if (kugel.getY()>219) {
75          return -5 * berechneEntfernung(kugel.getX(),kugel.getY(),schlaeger.getX(),schlaeger.getY());
76        }
77        else return 0;
78    }
79  }
```

Praktische Hilfsmethoden

In der Klasse AbstractGameWorld gibt es Hilfsmethoden, die man in den Unterklassen, vermutlich beim Setup, aufrufen kann:

- public final void setExplorationRate(double e)
- public void setBreakPeriodically(int i)
  Alle i Siege pausiert der Ablauf. Zur Diagnose einsetzbar.
- public final void setDisplayChanges(boolean b)
  Sollen Änderungen in den Tabellenwerten angezeigt werden? Ist schön, dauert Zeit.
- public final void setDisplayNextMoves(boolean b)
  Soll der nächste Zug angezeigt werden? Ist schön, dauert Zeit.
- public final void setDisplayWinsLossesStates(boolean b)
  Sollen die Anzahl von Siegen, Niederlagen und verrwalteten Zuständen angezeigt werden?
- public final void setStopUpdating(boolean b)
  Soll das Lernen ab jetzt aufhören, um einen bestimmten Zustand zu bewahren?
- protected void setUpdateOnGameEndOnly()
  protected void setUpdateOnEveryMove()
  Diese zwei Möglichkeiten schließen einander wechselseitig aus. Wird mit einem NeuralAgent gearbeitet, muss die zweite gewählt werden; das System versucht das selbständig zu erkennen.
- public final void setPlayers(Agent [] p)
  public final void setPlayers(Agent p)
  Muss aufgerufen werden, um einen oder ein Array von Agenten ins Spiel zu bringen. Die Agenten erhalten fortlaufende Nummern, sie werden der Reihe nach über die Zustände des Systems informiert (via getState), können dann Entscheidungen treffen, die sie via makeMove kommunizieren. Der Methode makeMove obliegt dann die Umsetzung in der Spielwelt.
- public void setVerbose(boolean b)
  Eine Hilfsmethode, die den Ausdruck auf der Konsole steuert. Jeder Agent hat insbesondere eine eigene, ebenso benannte Methode setVerbose.
- protected void showMessage(String s)

Agenten

Agent

Der normale Agent benutzt eine Q-Tabelle für das Reinforcement Learning. Der Konstruktor hat als Parametern die Explorationsrate, also 0.0 oder 0.05.

NeuralAgent

Dieser Agent benutzt ein Neuronales Netz für das Reinforcement Learning. Er erfordert das manuelle oder meist automatische Setzen setUpdateOnEveryMove() in der KI-Spiel-Klasse. Der Konstruktor des NeuralAgent hat vier Parameter:

- Erstens die Anzahl an Eingangsknoten – sie entsprechen den durch : getrennten Ganzzahlen, die den Zustand repräsentieren. Wenn getState das Format „1“ hat, ist das ein Eingangsknoten, beim Format „1:1“ sind es zwei, bei Werten wie „1:1:1:0:32“ wären es fünf.
- Zweitens die Anzahl an Knoten der einen versteckten Ebene. 10 ist eine gute Zahl.
- Drittens die Anzahl an Knoten der Ausgabeebene; sie muss der Anzahl an möglichen Zügen entsprechen; beim Breakout also 3 (für nichts, links, rechts), bei Snake 4 (für Nord, Ost, Süd, West).
- Viertens die Explorationsrate, wieder 0.0 oder 0.05, zum Beispiel.

Die Szenario-Unterklassen

Das sind reine Hilfsklassen, die das Arbeiten mit verschiedenen Szenarien erleichtern.

Die Auto-Szenarien

Das Autogame ist komplex. Es gibt zwei verschiedene Hintergründe, so dass man mit dem einen trainieren und das trainierte Modell dann am anderen Hintergrund testen kann. Ziel ist, dass ein Auto selbständig den Weg zum Ziel findet. (Mehrfachspielermodus ist nur in Ansätzen imoplementiert.) Das Spielfeld ist farbig kodiert, je dunkler die Farbe wird, desto näher ist man der Ziellinie aus der richtigen Richtung gekommen.

Der Zustand wird vom Fahrzeug-Objekt erzeugt. Es kann dabei zur Klasse Auto/SmoothAuto gehören, dann ist der Zustand sehr naiv und besteht aus x/y-Koordinaten: dann lernt das System sehr schnell, aber natürlich nur für diesen einen expliziten Kurs. Oder man nimmt die Klasse AutoSensor/SmoothAutoSensor: dann hat das Fahrzeug 3 oder 5 oder 7 (am besten erst einmal: 3) Sensoren, mit denen der Abstand nach vorne und zur Seite gemessen und als State verwendet wird. Das ganze geht außerdem als regulärer Actor oder als SmoothActor; letzteres ist eine beliebte Greenfoot-Lösung, um Rundungsfehler bei den x/y-Koordinaten zu reduzieren, die im Prinzip zu Winkel in Vielfachen von 45° führen.

Man kann darüber hinaus wie immer als Player einen Agent haben (Q-Table) oder einen NeuralAgent (Q-Net, nötig, weil der Zustandsraum bei dem Sensoren-Fahrzeug sehr groß ist).

Auch das Belohnungssystem ist komplex. Positive oder negative Belohnung kann es geben für: Vorwärtsfahren (eine dunklere Farbe erreicht, außer natürlich an der Startlinie), Rückwärtsfahren (hellere Farbe), Überqueren der Ziellinie, Verlassen der Fahrbahn.

Laden und Speichern: Geht mit Methoden der Klasse NeuralAgent.

Wie gut geht das alles?

Weniger gut, als man meint. Damit die KI bei Snake so gut spielt wie ein einfacher Algorithmus, muss die KI als Zustand nicht nur die Felder um sich herum kennen, oder die Anzahl freier Felder orthogonal zu ihr selbst, sondern im Idealfall das ganze Spielfeld im Blick haben – das sind mit 60 x 40 Eingangsknoten viele, und noch dazu rächt sich hier die Entscheidung, den Zustand als String zu modellieren und nicht gleich als Array von Zahlen.

Ginge das nicht auch einfacher?

Ja. Das Projekt ist halt entstanden aus bereits existierenden BlueJ-Projekten für Neuronale Netze und Reinforcement Learning, und greift außerdem auf bereits bestehende Greenfoot-Spieleprojekt zurück. Die Klassen PlayerManager und Game und so weiter, die nicht zur Greenfoot-Hierarchie gehören, kommen ursprünglich von BlueJ.

Was ist das mit dem Tic-Tac-Toe?

Lange Geschichte, der Versuch, das BlueJ-Projekt irgendwie zu visualisieren.

Fragen gerne an Thomas Rau, lehrerzimmer@herr-rau.de