Semesterbegleitende Leistung,

Teil II: Bestärkendes Lernen

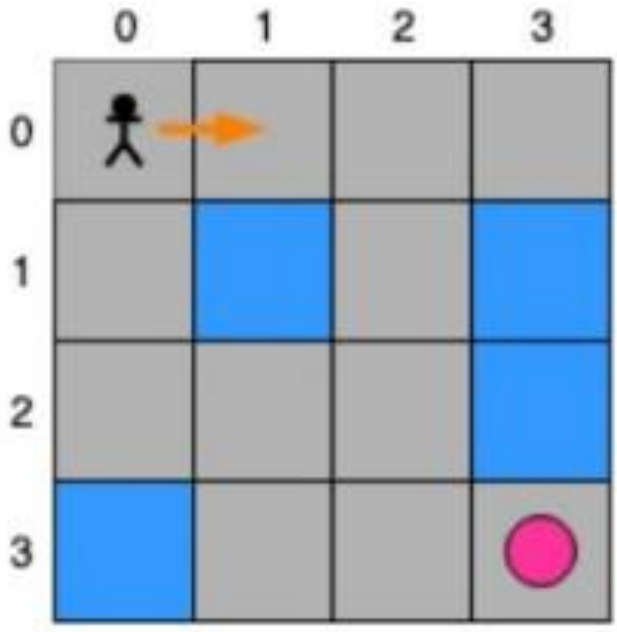
ALTERNATIVAUFGABE 2

Gruppenmitglieder: Henning Heidemann (201921854), Manuel Alberding (201824368), Bas Leon Seelig (201921609)

Inhalt

Problemstellung

Bei der zu bearbeitenden Problemstellung handelt es sich um das Frozenlake Problem. Dies soll mit Hilfe des Reinforcement Learing, zu Deutsch dem Bestärken Lernen umgesetzt werden. Bei dem Frozenlake Problem handelt es sich um die Überquerung eines Sees.



Der See wird über ein Raster dargestellt, bei dem es zwei Unterschiedliche Arten von Feldern gibt. Einerseits die grauen Felder welche starke Eisschichten darstellen, über die der Charakter laufen kann. Andererseits die blauen Felder, diese zeigen ein Loch in der Eisschicht auf. Wenn der Spieler auf eines dieser Felder tritt fällt er hinunter. Das Strichmännchen steht für den Spieler und der pinke Kreis ist die Frisbee, die er holen möchte.

Um das Problem zu lösen, soll das Reinforcement Learning verwendet werden, bei dieser Art der Programmierung soll das Programm anhand seiner Erfahrungen eine Strategie entwickeln, wie er das Problem lösen kann. Beim Start des Lernprozesses ist das System

unerfahren und kann nicht zwischen guten und schlechten Aktionen entscheiden. Das Problem soll nicht über die Erstellung einer Karte gelöst werden, sondern durch die Methoden des Reinforcement Learning. Am Ende soll das Programm erfolgreich einen möglichst kurzen Weg vom Start zum Ziel finden.

Reinforcement Learning

Reinforcement Learning ist neben überwachtem und unüberwachtem Lernen eine der drei grundlegenden Lernmethoden des maschinellen Lernens. Reinforcement Learning erfordert keine vorherigen Daten, sondern generiert Lösungen und Strategien basierend auf den erhaltenen Belohnungen. Es findet Lösungen und Strategien für Probleme basierend auf dem Trial-and-Error-Prozess und erhält Belohnungen für die Durchführung bestimmter Operationen. Im Vergleich zu anderen Lernmethoden ist zum Trainieren des lernenden Systems (Agent) kein Basisdatenmaterial erforderlich. Wissen und Intelligenz werden in vielen verschiedenen Simulationsläufen generiert. Der verwendete Algorithmus ist darauf ausgelegt, die erhaltenen Belohnungen zu maximieren. Es gibt keine spezifischen Regeln für einzelne Aktionen, sondern werden durch die Vorteile bestimmt, die durch die erhaltenen Belohnungen generiert werden.

Reinforcement Learning ist dem menschlichen Lernen sehr ähnlich und nutzt beispielsweise künstliche neuronale Netzwerke. Ein sehr bekanntes Beispiel für den Einsatz von Reinforcement Learning ist AlphaGo von Google. AlphaGo Zero kann sich mit den besten Go-Spielern der Welt messen und das Spiel selbst ohne menschliches Zutun erlernen.

Funktion

Für das Reinforcement Learning können verschiedene Algorithmen verwendet werden. Grundsätzlich basieren diese Algorithmen auf dem gleichen Prinzip. Die Aktionen des Agenten ändern die Systemumgebung. Beim Reinforcement Learning hat der Agent in der Ausgangssituation keine Informationen darüber, wie sich Aktionen auf die Systemumgebung auswirken. Je nachdem, ob die Veränderung im Problemlösungssinn positiv oder negativ ist, erhält der Agent Feedback in Form von Belohnung oder Nichtbelohnung. In diesem Fall geht es bei den Belohnungen um das nicht Herunterfallen also wenn der Agent auf einem grauen Feld steht. Die nicht Belohnung ist das Herunterfallen, also wenn der Agent auf ein blaues Feld tritt. Basierend auf dem erhaltenen Feedback unternimmt der Agent dann den nächsten Schritt. Das Ziel des Algorithmus ist es, die im Simulationssystem erhaltenen Belohnungen zu maximieren. Dies kann Konsequenzen haben, die zu Maßnahmen und Strategien zur Lösung des Problems führen. Das künstliche neuronale Netz bildet die Lernergebnisse in

seiner Neuronen Schicht ab. Die Problemlösung ist in den Neuronen zwischen dem Input- und Output-Layer gespeichert.

Neuronale Netze

Künstliche Neuronale Netze sind Algorithmen, die dem menschlichen Gehirn nachempfunden sind. Dieses abstrakte Modell der Verbindung künstlicher Neuronen ermöglicht es Computern, komplexe Aufgaben in den Bereichen Statistik, Informatik und Wirtschaftswissenschaften zu lösen. Neuronale Netze gelten als Grundlage der künstlichen Intelligenz. Mithilfe neuronaler Netze können verschiedene Datenquellen wie Bilder, Töne, Texte, Tabellen oder Zeitreihen interpretiert und Informationen oder Muster extrahiert werden, um sie auf unbekannte Daten anzuwenden. Auf diese Weise können datengetriebene Vorhersagen für die Zukunft getroffen werden. Künstliche neuronale Netze können unterschiedliche Komplexitätsgrade aufweisen, haben aber im Wesentlichen die Struktur eines gerichteten Graphen. Besitzt das künstliche neuronale Netz eine besonders tiefe Netzstruktur, spricht man von Deep Learning.

Deep Learning

Deep Learning beschreibt bestimmte Methoden des maschinellen Lernens, bei denen verschiedene Schichten einfacher Verarbeitungseinheiten mit dem Netzwerk verbunden werden. Wenn Sie Daten in das System einspeisen, durchlaufen sie nacheinander alle Schichten – genauso wie visuelle Informationen vom Auge über die Netzhaut zum Gehirn übertragen werden. Diese Tiefe ermöglicht es dem Netzwerk, komplexere Strukturen zu erlernen, ohne sich auf unrealistische große Datenmengen verlassen zu müssen.

Neuronale Netze beim beschränkten Lernen

Beim Reinforcement Learning weist der Lehrer für jedes Eingabemuster des Trainingssatzes nur darauf hin, ob die Klassifikation richtig oder falsch ist, nicht aber den richtigen Output. Das Netzwerk muss die richtige Ausgabe selbst finden. Es gibt jedoch keinen Zielwert für ein einzelnes Ausgabeneuron. Der Lernprozess selbst muss die richtige Ausgabe dieser Neuronen finden. Aus neurobiologischer oder evolutionärer Sicht ist diese Art von Lernprozess sinnvoller, da bei niederen und höheren Organismen einfache Rückkopplungsmechanismen in der Umwelt beobachtet werden können. Im Gegensatz dazu besteht das Problem bei diesen Lernmethoden darin, dass sie für Aufgaben mit bekannten erforderlichen Ausgaben länger brauchen als überwachte Lernmethoden, weil sie weniger Informationen für die richtige Gewichtsmodifikation haben.

Implementierung in Java

StateValue mit On-/Off-Policy

Implementierung bestärkendes Lernen mit StateValue Funktion und on/off – Policy

Vorab einmal: mit Hilfe der StateValue Funktion werden für die einzelnen Positionen auf dem Feld Werte berechnet um für den Lernenden Spieler ein Anhaltspunkt zu haben, wo dieser sich hinbewegen sollte und wohin nicht. On und Off Policy bezieht sich darauf wie der Spieler während des Lernens vorgeht. Wird die Off-Policy Strategie verwendet, bewegt wird sich der Spieler in eine zufällige Richtung bewegen. Bei On-Policy schaut der Spieler zuerst nach welches der verfügbaren nächsten Felder den höchsten Wert über die StateValue Funktion bekommen hat und bewegt sich anschließend dort hin. Die On-Policy Strategie erweist sich als, besser da bei dieser der Spieler sich in „sicheren“ Umgebungen bewegt und eher dazu tendiert nicht vor und zurück zu gehen. Da bei Off-Policy zufällige Positionen genutzt werden, könnte es dort auch passieren, dass der Spieler sich in einer Ecke aufhängt wo dieser nur im Kreis läuft und somit in der Lernphase das Ziel noch gar nicht erkunden konnte.

```
        while (currentEpoch < epochs) {
          // update state value table
          ArrayList<Tuple<Richtung, Koordinate>> possible = getValidDirections(myPlayer);
          for (Tuple<Richtung, Koordinate> tuple : possible) {
              Koordinate coord = tuple.second;
              double newValue = stateValue(coord, getReward(see.zustandAn(coord)), false);
              state_value[coord.getZeile()][coord.getSpalte()] = newValue;
          }

          // decide wether to use off or on policy
          // off-policy -> choose a random next position
          // on-policy -> choose the position with the currently highest value;
          if (!onPolicy) {
              int index = rnd.nextInt(possible.size());
              myPlayer = possible.get(index).second;
          } else {
              myPlayer = getMaxSequelPos(possible, false).second;
          }
          movesInEpoch++;

          // stop epoch and start over when:
          // - player got stuck in water
          // - player finished the maze
          // - number of moves in current epoch is bigger than allowed
          if (see.zustandAn(myPlayer) == Zustand.Wasser || see.zustandAn(myPlayer) == Zustand.UWasser
                  || see.zustandAn(myPlayer) == Zustand.Ziel || movesInEpoch > maxNumberMovesPerEpoch()) {
              currentEpoch++;
              movesInEpoch = 0;
              versuchZuende(see.zustandAn(myPlayer));// reset players position to the start
              System.out.println("\u001B[31mfinished epoch:\u001B[32m" + currentEpoch + "\u001B[0m");
          }
      }
```

Dies ist der Lernprozess des Spielers mit der StateValue Funktion.

Zu Beginn werden alle möglichen Richtungen des Spielers mithilfe der `getValidDirection()` Funktion ermittelt. Die Funktion gibt eine Liste von den erlaubten `Richtungen` und deren `Koordinate` in der diese Bewegung hin resultieren würde. Für alle Elemente der Liste wird dann der Wert an der korrekten Position im Array aktualisiert.

```
private double[][] state_value = new double[lake_size][lake_size];
```

das Array ist so definiert das es jede Position im Feld darstellen kann. Als erster index wird die Zeile angegeben und als zweiter die Spalte. Resultierend bekommt man daraus dann den momentanen Wert.

```
    private double stateValue(Koordinate player, double reward, boolean neuralnet) {
        double current = state_value[player.getZeile()][player.getSpalte()];
        double max_sequel = getMaxSequel(player, neuralnet);
        return (1 - learnrate) * current + reward * diskont + diskont * learnrate * max_sequel;
    }
```

Um den neuen Wert zu ermittlen wurde die aus der Vorlesung bekannte Formel:

$$V(s) = (1 - learnrate) * V(s) + reward * discount + learnrate * discount * maxSequelVal$$

Der *reward* ermittelt sich daraus welchen Zustand das betroffene Feld hat:

Eis --> -0.00000001 (sehr neutral da dies eine neutrale Bewegung ist)
Wasser --> -1 (schlecht, der Spieler hat an dieser Stelle verloren)
Ziel --> 0.5 (gut, der Spieler ist am Ziel angekommen)

Wir haben beim Testen mehrere Kombinationen von werten ausprobiert, allerdings erwiesen sich diese als am besten funktionierend aus

Die Funktion `getMaxSequel()` geht lediglich durch die möglichen Nachfolger einmal durch und ermittelt welcher von diesen den größten Wert im Array *state_value* stehen hat.

Nachdem die benötigten Parameter für die obige Formel bereit stehen wird der resultierende Wert zurückgegeben.

Wurden nun alle state_value Werte aktualisiert (*siehe Zeile 14 im Lernprozess*) wird geschaut ob der Spieler über On-Policy oder Off-Policy lernen soll. Ist Off-Policy gewählt, wird eine zufällige Position aus den verfügbaren ausgewählt. Ist Soll mit On-Policy gelernt werden, wird die Position gewählt, die den höchsten Wert hat. Auf diese Position wird der Spieler gesetzt und läuft von dort aus weiter.

Ein Lern-durchlauf geht solange bis:

- Der Spieler auf Wasser gestoßen ist
- Der Spieler das Ziel erreicht hat.
- Die Anzahl der bereits getätigten Schritte größer ist wie das festgelegt Maximum (SeeGröße*3)

Der Spieler wird an der Stelle wieder an die Start-Position gesetzt und die nächste Episode des Lernprozesses wird gestartet. Dies wird solange durchgeführt bis eine Anzahl von (in unserem Fall) 1000 Episoden erreicht wurde. Der Wert von 1000 Episoden hat durch ausprobieren als guter Wert für uns erwiesen.

StateValue mit Neuronalem Netz

Implementierung der StateValue Funktion mithilfe eines neuronalen Netzes

```
        int input_size = lake_size * lake_size;
        int output_size = 1;
        mlp = new MultiLayerPerceptron(TransferFunctionType.TANH,
                                              input_size, input_size, input_size, output_size);
        trainingSet = new DataSet(input_size, output_size);
```

Um mit einem neuronalen Netz zu arbeiten haben wir die Bibliothek neuroph benutzt.

Zur Erstellung des Netzes benutzen wir ein MultiLayerPerceptron mit der Transfer Funktion ***TANH***, einem Input Layer mit der Größe des Feldes, zwei Hidden Layer mit jeweils der Größe des Feldes und einem Output Layer der Größe 1. Die Größe der Input ist die Feldgröße damit wir als Input jede beliebige Position im Feld nehmen können. Die Anzahl von zwei Hidden Layer kommt dabei wieder durchs ausprobieren und hat für unseren Fall am besten funktioniert. Die Transfer Funktion ***TANH*** gibt uns einen Wertebereich von -1 bis 1 was für unseren Fall ziemlich gut passt.

*x läuft gegen **-1** -> schlechte Situation*

*x läuft gegen **+1** -> gute Situation*

Unser Trainingsdatensatz hat dementsprechend die korrespondierenden Parameter zu dem neuronalen Netz.

```
while (currentEpoch < epochs) {
    ArrayList<Tuple<Richtung, Koordinate>> possible = getValidDirections(myPlayer);
    for (Tuple<Richtung, Koordinate> tuple : possible) {
        Koordinate coord = tuple.second;
        double newValue = stateValue(coord, getReward(see.zustandAn(coord)), true);
        trainingSet.add(new DataSetRow(createInputVector(coord), new double[] { newValue }));
    }
```

Um nun mithilfe des neuronalen Netzes im Lernprozess zu lernen mussten wir die erfassten Werte in unseren Trainingsdatensatz hinzufügen. Um den neuen StateValue Wert zu ermitteln müssen wir nun allerdings die Werte aus unserem neuronalen Netzwerk benutzen.

```
    private double getStateValueNN(Koordinate player) {
        mlp.setInput(createInputVector(player));
        mlp.calculate();
        return mlp.getOutput()[0];
    }

    private double[] createInputVector(Koordinate player) {
        double[] input = new double[brd_size * brd_size];
        input[convert2DTo1D(player)] = 1;
        return input;
    }

    private int convert2DTo1D(Koordinate pos) {
        return brd_size * pos.getSpalte() + pos.getZeile();
    }
```

Um einen Wert aus dem neuronalen Netz zu berechnen erstellen wir zunächst einen Input Vektor basierend auf der gegebenen Position. Die Berechnung des neuen stateValue wertes für eine Position ist sonst gleich wie die auf der Datenstruktur aufgebaute Variante, nur das jetzt hier die Werte aus dem Neuronalen Netz gezogen werden. Nachdem die Werte zu dem Trainingsdatensatz hinzugefügt wurde, wird das Netz auf die neu gewonnen Werte gelernt.

```
    private void learnNN(DataSet set) {
        mlp.learn(set);
    }
```

Unsere Implementierung mithilfe des neuronalen Netzes funktioniert allerdings nur bis zu einer See-Größe von 6x6. Wird ein größerer See gewählt hängt sich der Spieler meist an einer Position auf und läuft an dieser vor und zurück. Vermutlich liegt das daran, dass evtl. die Parameter zur Berechnung nicht perfekt abgestimmt sind oder die Belohnungen für die Zustände angepasst werden müssen.

Zusatzaufgabe Q-Matrix

Zur Implementierung Bestärkendes Lernen mithilfe der Q-Matrix benötigen wir zur Speicherung der Ergebnisse eine Datenstruktur welche uns an einer Position(State) zu den möglichen Richtungen(Action) sagen kann wie „gut“ oder „schlecht“ eine Aktion an der Stelle ist.

```
private double[][][] q_matrix = new double[brd_size][brd_size][4];
```

Um das im Programm zu realisieren haben wir ein 3-dimensionales Array benutzt. Über die ersten beiden Indizes wird die Position angegeben(Zeile, Spalte) und der dritte Index beschreibt in welche Richtung sich bewegt wird.

```
            var possible = getValidDirectionsOnly(myPlayer);
            for (var t : possible) {
                var resultsInPosition = addDirToPos(myPlayer, t);
                var stateAction = new Tuple<Richtung, Koordinate>(t, myPlayer);
                double newValue = qValue(stateAction, getReward(see.zustandAn(resultsInPosition)), false);
                q_matrix[myPlayer.getZeile()][myPlayer.getSpalte()][t.ordinal()] = newValue;
            }

            if (!onPolicy) {
                int index = rnd.nextInt(possible.size());
                var dir = possible.get(index);
                var resultsInPosition = addDirToPos(myPlayer, dir);
                myPlayer = resultsInPosition;
            } else {
                var stateAction = getMaxValueQMatrixPos(myPlayer, false);
                myPlayer = addDirToPos(myPlayer, stateAction.first);// move to the best position
            }
```

Zum Lernen werden nun an der Position des Spielers Alle möglichen Richtungen rausgesucht in welche dieser sich bewegen darf. Für die möglichen Aktionen wird dann

anschließend der neue Q-Wert für diese Aktion berechnet und an der korrekten Stelle im Array *q_matrix* eingetragen.

Die Formel zur Berechnung des Q-Werts ist identisch zu der der StateValue Formel, nur das hier natürlich als Referenzwerte das *q_matrix* Feld benutzt wird.

Wurden alle Werte in der *q_matrix* aktualisiert wird wieder überprüft ob mit On- oder Off-Policy gelernt werden soll. Mit Off-Policy wird eine zufällige Aktion durchgeführt und bei On-Policy bewegt sich der Spieler in die Richtung mit dem bereits größten Wert in der *q_matrix*.

Dieser Vorgang geht dann wieder solange bis der Spieler in das Wasser gefallen ist, das Ziel erreicht hat oder die maximale Zugzahl erreicht hat. Danach wird der Spieler wieder an die Start-Position gestellt und beginnt von vorne.

Schrittauswahl nach beendeten Lernprozess

```
public Richtung naechsterSchritt(Zustand ausgangszustand) {
   if (!useStateValue) {
      var stateAction = getMaxValueQMatrixPos(myPlayer, useNeuralNet);
      myPlayer = addDirToPos(myPlayer, stateAction.first);
      return stateAction.first;
   }
   Tuple<Richtung, Koordinate> direction = getMaxSequelPos(myPlayer, useNeuralNet);
   myPlayer = direction.second;
   return direction.first;
}
```

Um nach dem Lernprozess die richtigen Schritte auszuwählen wird jeweils die/der Aktion(Q-Matrix)/Nachfolger(StateValue) ausgewählt, welche/r den höchsten Wert hat.

Probleme/Lösungen/Erweiterungen

Die Implementierungen des bestärkenden Lernens für StateValue und die Q-Matrix ohne das neuronale Netz ging eigentlich ziemlich schnell, da man sich ganz gut an den Erklärungsvideos auf Youtube entlanghangeln konnte, um die Vorgehensweise auf das Frozenlake Problem zu übertragen.

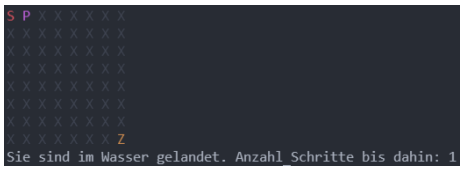
Dort mussten wir dann nur durch ein wenig rumprobieren die richtigen Parameter für die Berechnung und Anzahl der Episoden herausfinden.

Bei der Implementierung des neuronalen Netzes mussten wir uns erst einmal einlesen wie das genau funktioniert und dort auch recherchieren mit welchen Parametern wir das Neuronale Netz aufsetzen sollen. Da werden vermutlich auch noch Fehler drin sein, da das ja bisher nur bis zur Feldgröße von 6x6 funktioniert.

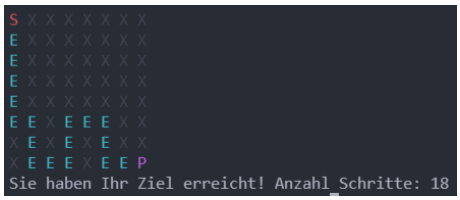
Anhang zur Funktionsweise

Genutzter See: „**Testsee8**“ aus den Beispielen die in Moodle hochgeladen sind

Ohne den Spieler anzulernen Lernen:



Nach dem Lernen:



Man erkennt das der Spieler ohne den Lernvorgang keinerlei Idee hat wo er überhaupt hinlaufen darf und daher in die erstbeste Möglichkeit läuft und sofort im Wasser landet.

Nach dem Lernen weiß der Spieler wo er hinlaufen darf und wohin nicht. Dadurch kann er das Ziel erreichen.

Ich habe die Ausgabe des Sees farblich so angepasst, damit es zum Testen besser zu sehen ist, welchen Weg der Spieler läuft.

Anbei auch noch einmal ein Ergebnis mit einem zufällig generierten See der Größe 40