MyProject Documentation

Contents

Module diceGameOptimizing

Implementiert das Reinforcement Learning als Optimierungsalgorithmus.

Sub-modules

- diceGameOptimizing.diceGame
- diceGameOptimizing.diceGameOptimizer
- diceGameOptimizing.evolutionary
- diceGameOptimizing.output
- diceGameOptimizing.reinforcement

Module diceGameOptimizing.diceGame

Classes

Class Game

```
class Game(
    pips,
    sides,
    rewardDraw=0.5
)
```

Diese Klasse implementiert das Würfelspiel, für welches Strategien optimiert werden sollen. Jede Interaktion mit dem Würfelspiel läuft über eine Instanz dieser Klasse.

Erstellt eine Instanz des Würfelspiels (pips, sides).

Parameters

pips : int Anzahl der zu verteilenden Augen
sides : int Anzahl der Seiten des Würfels
rewardDraw : float Auszahlung für ein Unentschieden

Methods

Method finished

```
def finished(
    self
)
```

Gibt an, ob ein angefangenes Spiel beendet wurde. Sollte nicht ohne gestartetes Spiel aufgerufen werden.

Returns

Ob das angefangene Spiel beendet wurde.

Method generateDice

```
def generateDice(
    self,
    usedSides,
    dice
)
```

Hilfsfunktion. Generiert alle möglichen Würfel inklusive Permutationen in DICE.

Method illegal

```
def illegal(
    self
)
```

Hilfsfunktion.

Method reward

```
def reward(
    self,
    output=False
)
```

Berechnet unter Beachtung des Wertes “rewardDraw” die Auszahlung eines abgeschlossenen legalen Spiels für den Spieler.

Returns

`reward` : int Gesamtauszahlung.

Method `start`

```
def start(
    self,
    playInOrder=False,
    invisibleGame=False
)
```

Startet ein zufälliges Spiel, außer wenn playInOrder True ist. In diesem Fall werden alle möglichen Würfel (inkl. Permutationen) nacheinander gespielt.

Parameters

`playInOrder` : bool Legt fest, ob alle Würfel der Reihe nach statt zufällig gespielt werden sollen.
`invisibleGame` : bool Legt fest, ob das gestartete Spiel mitgezählt wird.

Returns

Liste mit erstem Spielzug.

Method `takeAction`

```
def takeAction(
    self,
    action: int,
    output=False
)
```

Spielt ein bereits gestartetes Spiel weiter.

Parameters

`action` : int Zu verteilende Augenzahl.

Returns

`oppDice` : [int] Würfel des Gegners bisher.
`reward` : float Auszahlung für den Zug. Immer 0 außer bei letztem.

Module `diceGameOptimizing.diceGameOptimizer`

Functions

Function `optimizeDiceGame`

```
def optimizeDiceGame(
    pips,
    sides,
    optimizerType,
    newname=None,
    evo_kwargs=None,
    rl_kwargs=None,
    output=False
)
```

Hauptfunktion, die je nach Parametern eine Strategie optimiert.

Parameters

pips : int Augenanzahl für das Spiel.
sides : int Seitenanzahl für das Spiel.
optimizerType : str Legt Algorithmustyp fest; Entweder “EVO” oder “RL”.
newname : str Name für das Diagramm.
evo_kwargs : dict Parameter für EVO.
rl_kwargs : dict Parameter für RL

Module diceGameOptimizing.evolutionary

Implementiert die Evolutionären Algorithmen als Optimierungsalgorithmen.

Sub-modules

- diceGameOptimizing.evolutionary.agent
- diceGameOptimizing.evolutionary.evolutionarySearch
- diceGameOptimizing.evolutionary.strategy

Module diceGameOptimizing.evolutionary.agent

Classes

Class Agent

```
class Agent(
    game,
    strategyHandler,
    strat,
    changeRate
)
```

Implementiert einen Agenten für das Evolutionäre Lernen.

Erstellt einen neuen Agenten.

Parameters

game : Game Die Spielinztanz, um die Fitness zu bestimmen.
strategyHandler : StrategyAbstact Handelt die Strategie des Agenten.
strat : Any Strategie, direkt als Liste o. Ä.
changeRate : float Wert, wie stark die Strategie mit jeder Mutation angepasst werden soll.

Methods

Method changedAgent

```
def changedAgent(
    self
)
```

Gibt einen veränderten Agenten zurück.

Returns

Neues Objekt eines veränderten Agenten.

Method evaluateFitness

```
def evaluateFitness(
    self,
    fitnessAgainst,
    invisibleGame=False,
```

```
        output=False
    )
```

Bestimmt die Fitness des Agenten.

Parameters

fitnessAgainst : float | str Anteil, gegen wie viele Würfel (Anzahl) von allen gespielt werden soll. Falls "all" wird gegen jeden genau einmal gespielt.
invisibleGame : bool Legt fest, ob das Spiel gezählt wird.

Module diceGameOptimizing.evolutionary.evolutionarySearch

Classes

Class EvolutionarySearch

```
class EvolutionarySearch(
    game,
    strategyRep=0,
    populationSize=100,
    chooseBest=0.2,
    changeRate=1,
    generations=50,
    gamesToPlay=None,
    fitnessAgainst='all',
    output=False
)
```

Implementiert den Evolutionären Algorithmus. Lässt sich durch die train() Methode ausführen.

Parameters

game : Game Environment, das optimiert wird.
strategyRep : int Art der Strategiedarstellung. 0 -> Liste; 1 -> Vektor; 2 -> Vektor; 3 -> NeuronalesNetz
populationSize : int Populationsgröße.
chooseBest : float Anteil der Population, auf dem die Mutationen für die nächste Generation basieren.
changeRate : float Wert, wie stark die Agenten verändert werden. (Änderungsrate)
generations : int Anzahl der Generationen, über die trainiert wird.
gamesToPlay : int Anzahl der Spiele, die zum Training gespielt werden dürfen.
fitnessAgainst : int | str Spezifiziert die Fitnessbestimmung. (s. Agent)

Note: Wenn gamesToPlay nicht None ist wird für die Anzahl der Spiele trainiert. Ansonsten nach Anzahl der Generationen.

Methods

Method nextGeneration

```
def nextGeneration(
    self
)
```

Erstellt die nächste Generation an Agenten.

- Von allen Agenten die Fitness bestimmen.
- Die Agenten anhand ihrer Fitness sortieren.
- Die besten <code>chooseBest</code> auswählen.
- Fortpflanezen und mit leichten Veränderungen in die nächste Generation.

Method `train`

```
def train(
    self
)
```

Führt den Trainingszyklus aus.

Returns

`rewardPoints` : [(int, float)] Punkte (insgesamt gespielte Spiele, erreichte Auszahlung)

Module `diceGameOptimizing.evolutionary.strategy`

Sub-modules

- diceGameOptimizing.evolutionary.strategy.strat
- diceGameOptimizing.evolutionary.strategy.stratList
- diceGameOptimizing.evolutionary.strategy.stratVec
- diceGameOptimizing.evolutionary.strategy.stratVecComplete
- diceGameOptimizing.evolutionary.strategy.stratVecDoubleLayer

Module `diceGameOptimizing.evolutionary.strategy.strat`

Classes

Class `StrategyAbstact`

```
class StrategyAbstact(
    pips,
    sides
)
```

Abstrakte Klasse die notwendige Methoden vorgibt. Wird verwendet um die einzelnen Arten der Repräsentation Strategien zu implementieren. Nur für Evolutionäre Suche.

Ancestors (in MRO)

- abc.ABC

Descendants

- diceGameOptimizing.evolutionary.strategy.stratList.StratList
- diceGameOptimizing.evolutionary.strategy.stratVec.StratVec
- diceGameOptimizing.evolutionary.strategy.stratVecComplete.StratVecComplete
- diceGameOptimizing.evolutionary.strategy.stratVecDoubleLayer.StratVecDoubleLayer

Methods

Method `changedStrategy`

```
def changedStrategy(
    self,
    stategy,
    changeRate
)
```

Passt eine gegebene Strategie an.

Parameters

`strategy` : Any Zu ändernde Strategie.
`changeRate` : float Änderungsrate für die Strategie.

Returns

Veränderte Strategie.

Method `nextMove`

```
def nextMove(
    self,
    opponentsMoves,
    strategy
)
```

Bestimmt aus gegnerischem Würfel und einer Strategie den nächsten Zug.

Parameters

`opponentsMoves` : [int] Alle bisherigen Züge des Gegners.
`strategy` : Any Strategie, nach der gespielt werden soll.

Returns

Augenzahl, die gespielt werden soll.

Method `randomStrategy`

```
def randomStrategy(
    self
)
```

Erstellt eine zufällige Strategie.

Returns

Eine Zufällige Strategie.

Module `diceGameOptimizing.evolutionary.strategy.stratList`

Classes

Class `StratList`

```
class StratList(
    pips,
    sides
)
```

Implementiert die Strategiedarstellung über Listen.

Initialisiert neuen Strategie Handler.

Ancestors (in MRO)

- diceGameOptimizing.evolutionary.strategy.strat.StrategyAbstact
- abc.ABC

Methods

Method `changedStrategy`

```
def changedStrategy(
    self,
    strategy,
    changeRate
)
```

Verändert eine gegebene Strategie und gibt eine neue zurück.

Method `createStates`

```
def createStates(
    self,
    usedSides,
    state
)
```

Hilfsfunktion. Erstellt alle States.

Method `nextMove`

```
def nextMove(
    self,
    opponentsMoves,
    strategy
)
```

Gibt den nächsten Zug zurück

Method `numOfState`

```
def numOfState(
    self,
    state
)
```

Hilfsfunktion. Gibt Nummer eines States an.

Method `randStrategyRecursive`

```
def randStrategyRecursive(
    self,
    usedSides,
    usedPips,
    strategy,
    strategy_usedPips,
    startingPlayer,
    alreadyUsedPips
)
```

Hilfsfunktion. Erstellt mit rekursivem Prüfen auf Legalität eine zufällige Strategie.

Method `randomStrategy`

```
def randomStrategy(
    self
)
```

Ruft das rekursive Erstellen einer Random Strategie auf und gibt diese zurück.

Method `traceState`

```
def traceState(
    self,
    stateNum,
    strategy
)
```

Schauf ob eine Strategie, die an einer bestimmten Stelle bearbeitet wurde legal ist.

Module diceGameOptimizing.evolutionary.strategy.stratVec

Classes

Class StratVec

```
class StratVec(
    pips,
    sides
)
```

Implementiert die Strategiedarstellung über einfache Vektoren.

Initialisiert neuen Strategie Handler für einfache Vektor-Strategien.

Ancestors (in MRO)

- diceGameOptimizing.evolutionary.strategy.strat.StrategyAbstact
- abc.ABC

Methods

Method changedStrategy

```
def changedStrategy(
    self,
    strategy,
    change
)
```

Gibt eine veränderte Strategie zurück.

Method convertToStratList

```
def convertToStratList(
    self,
    toConvert
)
```

Hilfsfunktion. Umwandlung zu einer Listen-Strategie.

Method nextMove

```
def nextMove(
    self,
    opponentsMoves,
    strategy
)
```

Gibt den nächsten Zug zurück.

Method randomStrategy

```
def randomStrategy(
    self
)
```

Erstellt eine zufällige Strategie.

Module diceGameOptimizing.evolutionary.strategy.stratVecComplete

Classes

Class StratVecComplete

```
class StratVecComplete(
    pips,
    sides
)
```

Implementiert die Strategiedarstellung über größere Vektoren.

Initialisiert neuen Strategie Handler für eine größere Vektor-Strategien.

Ancestors (in MRO)

- diceGameOptimizing.evolutionary.strategy.strat.StrategyAbstact
- abc.ABC

Methods

Method changedStrategy

```
def changedStrategy(
    self,
    strategy,
    changeRate
)
```

Gibt eine veränderte Strategie zurück.

Method convertToStratList

```
def convertToStratList(
    self,
    toConvert
)
```

Hilfsfunktion. Umwandlung zu einer Listen-Strategie.

Method nextMove

```
def nextMove(
    self,
    opponentsMoves,
    strategy
)
```

Gibt den nächsten Zug zurück.

Method randomStrategy

```
def randomStrategy(
    self
)
```

Erstellt eine zufällige Strategie.

Module diceGameOptimizing.evolutionary.strategy.stratVecDoubleLayer

Classes

Class StratVecDoubleLayer

```
class StratVecDoubleLayer(
    pips,
    sides
)
```

Implementiert die Strategiedarstellung über einfache Vektoren.

Initialisiert neuen Strategie Handler für NN-Strategien.

Ancestors (in MRO)

- diceGameOptimizing.evolutionary.strategy.strat.StrategyAbstact
- abc.ABC

Static methods

Method activation

```
def activation(
    x
)
```

Hilfsfunktion. Aktivierungsfunktion Sigmoid.

Methods

Method changedStrategy

```
def changedStrategy(
    self,
    strategy,
    change
)
```

Verändert eine gegebene Strategie und gibt eine neue zurück.

Method convertToStratList

```
def convertToStratList(
    self,
    toConvert
)
```

Hilfsfunktion. Umwandlung zu einer Listen-Strategie.

Method nextMove

```
def nextMove(
    self,
    opponentsMoves,
    strategy
)
```

Gibt den nächsten Zug zurück

Method `randomStrategy`

```
def randomStrategy(
    self
)
```

Erstellt eine zufällige Strategie. **Note:** Inputlayer: (s-1)(p+1) Neuronen. Gewichte dazwischen: HIDDEN_LAYER_SIZE x (s-1)(p+1) Hiddenlayer: HIDDEN_LAYER_SIZE Gewichte dazwischen (p+1) x (HIDDEN_LAYER_SIZE) Outputlayer: (p+1) Neuronen

Module `diceGameOptimizing.output`

Functions

Function `evalSameNameTestRuns`

```
def evalSameNameTestRuns(
    testRunRewardPoints: [(<class 'str'>, <class 'float'>)]
)
```

Function `generatePlot`

```
def generatePlot(
    testRuns: (<class 'str'>, [(<class 'int'>, <class 'float'>)])
)
```

Generiert aus den Reward-Punkten vno verschiedenen Durchläufen der Algorithmen ein Diagramm. Sortiert verschiedene Durchläufe nach Name zusammen. Gibt Minimal-, Maximal-, Durchschnittswert an.

Parameters

`testRuns` **: (str, [(int, float)])**

Function `maxlists`

```
def maxlists(
    lists
)
```

Function `meanlists`

```
def meanlists(
    lists
)
```

Function `minlists`

```
def minlists(
    lists
)
```

Function `varlists`

```
def varlists(
    lists,
    add
)
```

Namespace diceGameOptimizing.reinforcement

Sub-modules

- diceGameOptimizing.reinforcement.agent
- diceGameOptimizing.reinforcement.reinforcementLearning
- diceGameOptimizing.reinforcement.strategy

Module diceGameOptimizing.reinforcement.agent

Classes

Class Agent

```
class Agent(
    game,
    strategy
)
```

Methods

Method evaluateFitness

```
def evaluateFitness(
    self
)
```

Method generationComplete

```
def generationComplete(
    self
)
```

Method nextMove

```
def nextMove(
    self,
    state
)
```

Method reinforce

```
def reinforce(
    self,
    state,
    nextState,
    action,
    reward
)
```

Module diceGameOptimizing.reinforcement.reinforcementLearning

Classes

Class ReinforcementLearning

```
class ReinforcementLearning(
    game,
    defaultQValue=0.5,
    strategyRep=0,
```

```
        alpha=0.9,
        endAlpha=None,
        gamma=0.5,
        epsilon=0.3,
        endEpsilon=None,
        epsilonDecay=0.999,
        timeSteps=100,
        rewardPointDensity=0.001,
        output=False
    )
```

Implementiert das Reinforcement Learning in Form von Q-Learning (ε-greedy). Lässt sich durch die train() Methode ausführen.

Parameters

`game` **: Game** Environment, das optimiert wird.
`defaultQValue` **: float** Standardwert im Q-Table
`strategyRep` **: int** Art der Darstellung des Q-Table
`alpha` **: float** konstanter Wert α für Q-Learning
`gamma` **: float** konstanter Wert γ für Q-Learning
`epsilon` **: float** Startwert des ε
`endEpsilon` **: float** Endwert des ε
`epsilonDecay` **: float** Faktor, mit dem ε nach jedem Spiel verringert wird
`timeSteps` **: int** Anzahl der zu spielenden Spiele
`rewardPointDensity` **: float** gibt die Dichte der rewardPoints an

Warning: Es muss 0 <= strategyRep <= 0 gelten.

Note: Bei Verwendung von endEpsilon wird epsilonDecay ignoriert!

Note: Bei zu großen Werten von defaultQValue wird stürzt der Algorithmus in eine Depression.

Methods

Method `nextGeneration`

```
    def nextGeneration(
        self
    )
```

Spielt ein Spiel und führt dabei das Q-Learning aus.

Method `train`

```
    def train(
        self
    )
```

Führt Trainingszyklus über timeSteps viele Spiele aus.

Returns

`rewardPoints` **: [(int, float)]** Punkte (insgesamt gespielte Spiele, erreichte Auszahlung)

Module `diceGameOptimizing.reinforcement.strategy`

Diese Untermodul beinhaltet alle Darstellungen von Strategien (Policies) fürs Q-Learning.

Sub-modules

- diceGameOptimizing.reinforcement.strategy.stratQTable

Module diceGameOptimizing.reinforcement.strategy.stratQTable

Functions

Function argNmax

```
def argNmax(
    a,
    N,
    axis=None
)
```

Classes

Class StratQTable

```
class StratQTable(
    game,
    defaultQValue,
    alpha,
    alphaDecay,
    gamma,
    epsilon,
    epsilonDecay
)
```

Implementiert eine Q-Table auf Basis eines dict.

Parameters

game : Game Environment, das optimiert wird.
defaultQValue : float Standardwert im Q-Table.
strategyRep : int Art der Darstellung des Q-Table.
alpha : float Konstanter Wert α für Q-Learning.
alphaDecay : float Faktor, mit dem α nach jedem Spiel verringert wird.
gamma : float Konstanter Wert γ für Q-Learning.
epsilon : float Startwert des ε.
epsilonDecay : float Faktor, mit dem ε nach jedem Spiel verringert wird.

Methods

Method bestMove

```
def bestMove(
    self,
    state
)
```

Gibt den besten Zug zurück. (ohne ε-greedy)

Method generateQTable

```
def generateQTable(
    self,
    defaultQValue
)
```

Hilfsfunktion. Generiert alle keys für den Q-Table. Nicht nötig mit “lazy” Ansatz zur Erstellung.

Method `nextMove`

```
def nextMove(
    self,
    state
)
```

Wählt nach ε-greedy Strategie den nächsten Zug aus. Erweiterung: Die Aktion mit dem n. größten Wert wird zur Wahrscheinlichkeit (1-ε)^n ausgewählt.

Method `nthBestMove`

```
def nthBestMove(
    self,
    state,
    nth
)
```

Rekursive Funktion, um mit Wahrscheinlichkeit ε nicht den nth Zug zu wählen.

Method `printqtable`

```
def printqtable(
    self
)
```

Hilfsfunktion. Ausgabe des Q-Table.

Method `reinforce`

```
def reinforce(
    self,
    state,
    nextState,
    action,
    reward
)
```

Update der Q-Funktion.

Parameters

`state` : ((int,...),(int,...)) Zustand.
`nextState` : ((int,...),(int,...)) Nächster Zustand.
`action` : int Gewählte Aktion.
`reward` : float Erlangte Belohnung.

Generated by *pdoc* 0.9.2 (https://pdoc3.github.io).

PDF-ready markdown written to standard output. ^^^^^^^^^^^^^^^^ Convert this file to PDF using e.g. Pandoc:

```
pandoc --metadata=title:"MyProject Documentation"              \
       --from=markdown+abbreviations+tex_math_single_backslash  \
       --pdf-engine=xelatex --variable=mainfont:"DejaVu Sans"   \
       --toc --toc-depth=4 --output=pdf.pdf  pdf.md
```

or using Python-Markdown and Chrome/Chromium/WkHtmlToPDF:

```
markdown_py --extension=meta         \
            --extension=abbr         \
            --extension=attr_list    \
            --extension=def_list     \
            --extension=fenced_code  \
```

```
            --extension=footnotes    \
            --extension=tables       \
            --extension=admonition   \
            --extension=smarty       \
            --extension=toc          \
            pdf.md > pdf.html

chromium --headless --disable-gpu --print-to-pdf=pdf.pdf pdf.html

wkhtmltopdf --encoding utf8 -s A4 --print-media-type pdf.html pdf.pdf
```

or similar, at your own discretion.