Leandro Conte

BWINF Runde 2 Aufgabe 1

Erweiterung: Heuristisches Verfahren mit neuronalem Netz

Idee

Das Standard-Lösungsverfahren, dass in der Dokumentation vorgestellt wird, versucht in mehreren Zyklen die Anordnung durch das Einsetzen neuer Rechtecke zu verbessern. Dieses Verfahren kann in „randomisierter“ Form mehrmals wiederholt werden, um eine größere Menge an Anordnungen zu erhalten, aus der dann die Beste gewählt werden kann.

Ein verbesserungsfähiger Punkt dieser Lösung ist die Laufzeit, welche zwar polynomial ist, vorausgesetzt die Zyklenzahl ist beschränkt, in Praxis jedoch mehrere Sekunden für eine einfache Lösung braucht.

Also habe ich mir, um einzelne Entscheidungen im heuristischen Verfahren zu optimieren, genauer genommen die anfängliche Sortierung der Rechtecke und die Ordnung in der die möglichen Platzierungen ausprobiert werden, eine neues Verfahren überlegt. Dieses verwendet zwei neuronale Netzwerke: das „ordering“ Netzwerk, dass die Rechtecke sortiert und das „placement“ Netzwerk, dass die Platzierung bestimmt. Diese Netzwerke werden in mehreren Generationen trainiert.

Anmerkung: Die hier vorgestellte Erweiterung ist ein Experiment und erkundet eine Reihe an Konzepten ohne sie perfekt einzusetzen.

Das Neuronale Netzwerk

Im Rest der Beschreibung werde ich als Platzhalter ON für das ordering Netzwerk und PN für das placement Netzwerk verwenden.

Das ON erhält als Eingabe (pro sample) einen Tensor der Größe (MAX_N, 6). MAX_N ist das größte N das verarbeitet werden kann. In der 2. Dimension befinden sich für jedes Rechteck:

Die Identifikationszahl local_ID
Die Anfangsstunde
Die Endstunde
Die Meterlänge
Der Anfangsmeter
Der Endmeter

Falls ein Wert nicht verfügbar ist, weil ein Rechteck noch nicht platziert wurde, oder weil N < MAX_N, wird -1 eingesetzt.

Das PN erhält noch einen extra Eintrag, das aktuelle Rechteck, für das die Platzierung gesucht wird.

ON gibt einen Tensor der Größe MAX_N zurück, eine Wahrscheinlichkeitsverteilung nach der die Rechtecke sortiert werden.

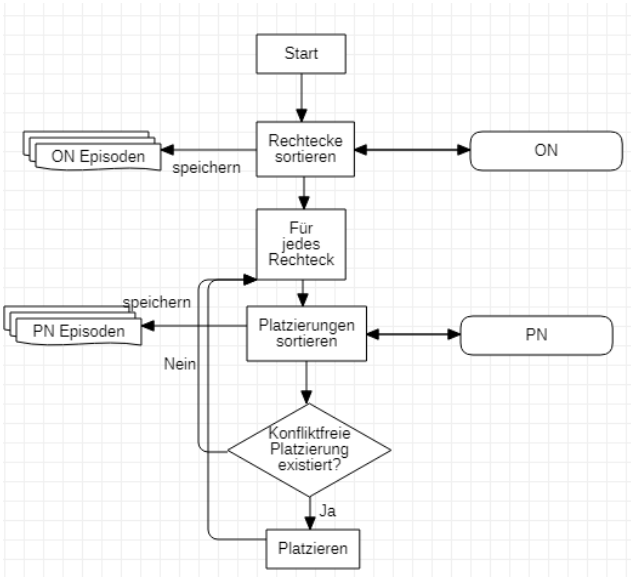
PN hingegen gibt eine Verteilung der Größe TLEN +1 zurück, nach der die Platzierungen sortiert werden. Die extra Platzierung ist für den Fall, dass das Rechteck nicht eingesetzt werden kann.

Beide Netzwerke enthalten 3 *Dense* Schichten. Schichten 1-2 benutzen „relu“ als Aktivierungfunktion, während die letzte Schicht „sigmoid“ für die Verteilung verwendet. Der Fehler wird für beide Netzwerke mit MSE berechnet.

Beide Netzwerke sind sehr generisch und könnten noch optimiert werden. Der Kernpunkt dieser Erweiterung ist nicht das Design der Netzwerke, da ich darin zu wenig Erfahrung habe, sondern vielmehr die Entwicklung der neuen angepassten Verarbeitungsfunktion und eines Trainingsvorgangs und das Experimentieren mit dem Potential, das diese komplexere Heuristik hat.

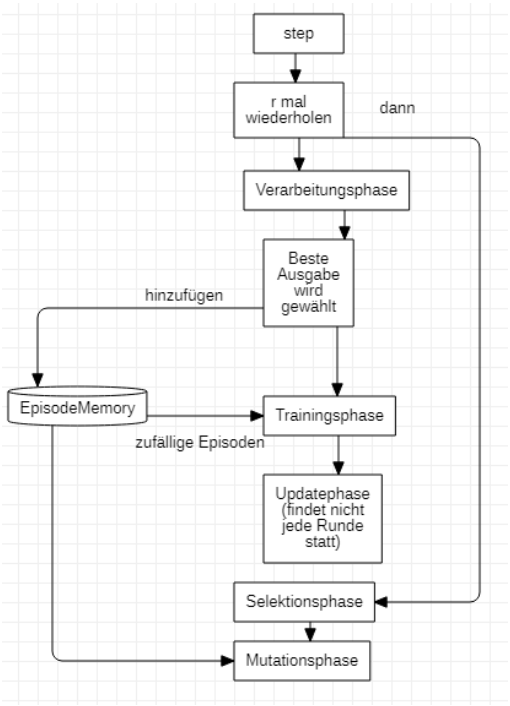
Die neue Verarbeitungsfunktion

Die neue Funktion *process_guided* läuft nicht mehr in Zyklen, sondern versucht jedes Rechteck nur einmal einzusetzen. Einmal gesetzt wird ein Rechteck nicht entfernt. Es werden weiterhin die Container aus aufgabe1_common.py verwendet.



Zusätzlich zu *TaskInput* benötigt *process_guided* zwei Funktionen, die als wrapper um die Netzwerke funktionieren. In der Abbildung rechts sieht man, wie die Verarbeitung abläuft. Jede Abfrage eines Netzwerks wird als Episode gespeichert. Jede Episode enthält die Netzwerk Eingabe und korrigierte Ausgabe. Wenn die Ausgabe nämlich einen Wert enthielt, der absolut sicher anders sein müsste, wird er korrigiert. Diese Episoden werden während dem Training gebraucht. Um die Eingabe für die Netzwerke nicht jedes Mal neu zusammenstellen zu müssen, wird die aktuelle Version in *reqArray* erhalten. Die erhaltenen Netzwerk wrapper werden von noch einmal zwei wrappern *reqOrdered* und *placeOrdered* benutzt, die die Ausgabe der Netzwerke konvertieren. Die Funktionen, die Konflikte finden *findInSet* und *findInArray*, geben hier nur einen Wahrheitswert zurück und keine Liste an kollidierenden Rechtecken. Am Ende der Verarbeitung wird das Ergebnis zusammen mit den gespeicherten Episoden in einem *TaskOutputGuided* Objekt ausgegeben.

Der Trainingsvorgang

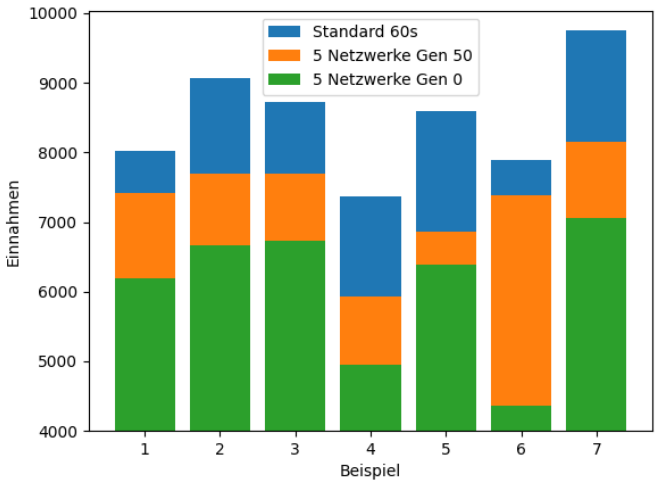


Die Netzwerke werden in mehreren Generationen trainiert. Der benutzte Vorgang ist eine Art Deep Learning und folgt einer Mischung an Strategien. Ein aktueller Trainingszustand wird in *TrainingState* Objekten enthalten. Diese Klasse enthält alle Parameter, Methoden und Netzwerke des Trainings, sowie eine Möglichkeit den Zustand in einem Ordner zu speichern. Eine „Generation“ enthält eine bestimmte Anzahl an Netzen. Das Problem der gestellten Aufgabe, Rechtecke anzuordnen, ist, dass es keinen direkten Weg gibt die Qualität einer Lösung zu beurteilen. Stattdessen wird die Qualität der Netzwerke und Antworten durch Vergleiche bestimmt. Mit der *step* Methode (siehe Abbildung) berechnet TrainingState die nächste Generation, den nächsten Trainingszustand. *step* enthält mehrere Phasen. In der

Verarbeitungsphase werden zufällige Eingaben verarbeitet. *rProb* ist die Wahrscheinlichkeit, dass der Verarbeitungfunktion eine zufällige Entscheidung gegeben wird, wobei das zugewiesene Netzwerk, das die Entscheidung hätte treffen sollen, übersprungen wird. Die Entscheidungen, die *process_guided* erhält, sind also vor allem am Anfang dank rProb teilweise zufällig sind, um mehr Diversität zu ermöglichen. Pro Eingabe werden die Episoden der besten Ergebnisse zu einem begrenzten „Pool“ an Episoden: *EpisodeMemory* hinzugefügt. Aus diesem werden zufällige Episoden ausgewählt, auf deren Basis die Netzwerke trainiert werden. Wichtig ist hier, dass es von einem Netzwerk immer zwei Versionen gibt: die Version die sofort trainiert wird und die, die die Entscheidungen an *process_guided* liefert. Alle x Wiederholungen wird die Entscheidungen liefernde Version aktualisiert. Nachdem die Netzwerke trainiert wurden werden in der Selektionsphase die Besten von ihnen ausgewählt. Der Rest der Netzwerke wird gelöscht. Um die gelöschten Netzwerke zu ersetzen werden die übrigen kopiert. In der „Mutationsphase“ wird die neue Menge an Netzwerke ein letztes mal mit zufälligen Episoden trainiert, damit die kopierten Netzwerke nicht exakt gleich, sondern leicht anders sind.

Ergebnis

Ich habe die Idee mit Hilfe der tensorflow und keras Module in Python implementiert und in Google Colab trainiert und getestet. Dank der freien GPUs in Colab konnte ich den Trainingsvorgang mehrmals testen und die Leistung über mehrere Generationen vergleichen. Dabei habe ich einzelne Vorgänge und Parameter optimiert. Trotzdem ist noch ein großer Raum für Verbesserungen übrig. Interessanterweise ist die aktuelle Version der Netzwerke leistungsfähiger als ich geschätzt habe. Nach 50 Generation, mit einem anfänglichen rProb=0.2, welches jede Generation verringert wurde, erzielt die neue Verarbeitungsfunktion mit den trainierten Netzwerken bei manchen Beispielen schon gute Resultate. Diese sind zwar noch schlechter, als die der Standard Methode, sind aber schon nah dran und auf jeden Fall besser als die untrainierten Resultate. Der Trainingsvorgang, wurde für 5 Netzwerke mit 4-6 Wiederholungen pro step ausgeführt. Das Training für die 5 Netzwerke hat ca. 45 Minuten gedauert und fand ausschließlich auf Basis von zufälligen Eingaben statt. Diese wurden abwechselnd mit dense=True und dense=False generiert. Am Schluss wurden das 1. Netzwerkpaar auf die 7 Beispiele angewandt. Die Ergebnisse, welche von checkNoCollisions überprüft wurden, sieht man rechts. Ich vermute, dass man die Leistungsfähigkeit dieser Methode, durch feineres Einstellen der viele Trainingsparameter und durch weiteres Training noch weiter verbessern kann. Unter dem Google-Drive Link unten findet man den Zustand bei Generation 90, der mit *save* gespeichert wurde.



https://drive.google.com/drive/folders/1Ma084pSfX9vnuxIqq81jPL_03jPm1uLS?usp=sharing