Praktikumsbericht: Implementierung tiefer Neuronaler Netze auf der GPU

Julien Aziz

Institut für Informationssicherheit und Verlässlichkeit (KASTEL)
Betreuender Mitarbeiter: Daniel Zimmermann

1 Einführung

Die Idee des Neuroveolutionäneren Ansatzes ist es, eine Anfangspopulation von Neuronalen Netzen, evolutionär zu optimieren um so performante Modelle zu generieren. Basis für diese Optimierung ist die Evaluation jedes Individuums innerhalb einer Trainingsumgebung. Eine solche Umgebung wäre zum Beispiel eine graphische Benutzeroberfläche. Dort traversiert ein Agent mit Klickoperationen durch die verschiedenen Zustände der Software, wobei die Aktionen des Agenten durch ein Neuronales Netz gesteuert werden. In jedem Evolutionsschritt werden die Ergebnisse innerhalb dieser Umgebung genutzt, um eine neue Generation an Neuronalen Netzen zu erzeugen [2]. Für diese Problemstruktur bieten sich Rekurrente Netze besonders an, da diese ein Art Gedächtnis implementieren und somit der Agent die Zustände in denen er bereits gewesen ist, in die Berechnung der nächsten Klickoperation miteinbeziehen kann. Der Nachteil dieser Art des Trainings ist die Anzahl an Interaktionen mit der Trainingssoftware. Im Allgemeinen ist die benötigte Anzahl an Interaktionen um eine gewisse Genauigkeit zu erreichen, deutlich höher als mit gängigen Reinforcement-Learning Ansätzen. Da sich die Struktur des Trainingsprozesses für eine umfassende Parallelisierung anbietet, kann dieser Nachteil durch einen hohen Parallelisierungsgrad der Implementierung umgangen werden.

Im Rahmen dieses Praktikums sollten die Rekurrente Netze:

1. Elman-Netz,
2. Gated reccurent unit (GRU),
3. Long short-term memory (LSTM),

in das NeuroEvolution Framework, unter der Programmiersprache Julia implementiert werden. Die Hauptfunktionalitäten der Netze sollten hierbei auf der GPU realisiert werden um die umfangreichen Parallelisierungsmöglichkeiten von Grafikkarten ausnutzen zu können.

2 Anforderungen

Die Anforderungen an die Neuronale Netze sind im wesentliche zwei Hauptfunktionalitäten:

1. Die korrekte Initialisierung der Modelle vor jeder Trainingsgeneration.
2. Korrekte Berechnung der Ausgabe zu gegebenen Eingaben über lange Zeithorizonte.

Die Korrektheit der Ausgaben soll dabei mithilfe der Machine-Learning Bibliothek "Flux"validiert werden.

2.1 Funktionale Anforderungen

Folgende funktionale Anforderungen wurden an die Implementierung gestellt

/FA10/: Die Netze speichern die nötigen Gewichtsmatritzen, interne Zustände, Anzahl der Neuronen sowie Größe der Ein- und Ausgaben.

/FA20/: Anzahl der Neuronen soll konfigurierbar sein.

/FA30/: Anzahl der Ein- und Ausgaben soll konfigurierbar sein.

/FA40/: Die Gewichtsmatritzen der Netze werden entsprechend der Ausgabe vom Optimierer korrekt initialisiert.

/FA50/: Die internen Zustände werden für jeden Zeitschritt korrekt berechnet.

/FA60/: Die Ausgabe wird für jeden Zeitschritt korrekt berechnet.

2.2 Nicht-Funktionale Anforderungen

/NFA10/: Initialisierung und Schrittfunktion der Umgebung sollen über CUDA auf der GPU ausgeführt werden.

/NFA20/: Die Netze sollen für jedes Individuum parallel erstellt werden.

/NFA30/: Die Ausgabe soll für jedes Individuum parallel berechnet werden.

/NFA40/: Die Initialisierung und Ausgabeberechnung innerhalb eines Individuums soll ebenfalls mittels Threads parallelisiert werden.

/NFA50/: Effiziente Ausnutzung des statischen, geteilten GPU-Speichers.

/NFA60/: Initialisierung und Schrittfunktion sind auf der GPU ausführbar.

/NFA70/: Die verwendeten Berechnungen der Neuronalen Netze sollen identisch mit denen in Flux verwendeten Varianten sein.

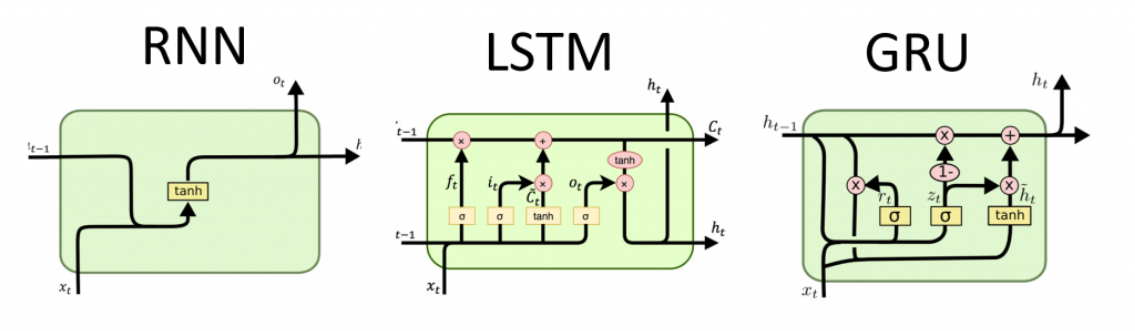

Abbildung 1: Visualisierung von Elman-Netze (RNN), Long short-term Memory (LSTM) und Gated recurrent unit (GRU), aus [1]

3 Rekurrente Neuronale Netze

In diesem Abschnitt werden die mathematischen Grundlagen der Rekurrenten Neuronalen Netze (RNN) behandelt und die implentierten Varianten präsentiert. Der fundamentale Unterschied zu Feed Forward Netzen ist, dass RNNs einen internen Zustand speichern und pflegen. Dieser interne Zustand wird innerhalb von versteckten Schichten mit den Eingaben verknüpft, um somit die Ausgabe abhängig vom aktuellen Kontext zu generieren. So wird ein Gedächtnis implementiert um Sequenzen von Eingaben kontextbasiert zu verarbeiten. Die Verknüpfungen der Eingaben mit den internen Zuständen ist der Kern der RNNs und hängt von der Art des betrachteten Modells ab.
In den Implementierten Varianten wird hinter die versteckte Schicht noch eine Ausgabeschicht geschaltet. Diese generiert auf Basis der Ausgabe der verstecken Schicht h_t, die Ausgabe des Netzes y_t zum Zeitpunkt t:

$$y_t = (W_y h_t + b_y),$$

wobei W_y und U_y Gewichtsmatrizen darstellen und b_y den Bias. Im Kontext der Trainingsumgebung stellt diese Ausgabe die nächste Aktion des Agenten dar.

3.1 Elman-Netz

Die einfachste Realisierung eines rekurrenten Neuronalen Netzes ist das Elman-Netz. Die Ausgabe h_t der versteckten Schicht zum Zeitschritt t wird hierbei durch eine einfache tangens hyperbolicus Verknüpfung der Eingabe x_t mit dem vorherigen internen Zustand h_{t-1} berechnet:

$$h_t = tanh(W_h x_t + U_h h_{t-1} + b_h).$$

Eine Visualisierung des Ablaufs, sowie die der anderen Netze kann in Abbildung1 betrachtet werden.

3.2 Gated Recurrent Unit (GRU)

Gated reccurent units nutzen sogenannte *Gates* um die alten Informationen aus dem letzten internen Zustand h_{t-1} selektiv zu verarbeiten und als Filter für die Ausgabe h_t.

GRUs verwenden ein *Resetgate* und ein *Updategate* um den nächsten internen Zustand h_t zu berechnen,

$$
\begin{aligned}
r_t &= \sigma(W_r x_t + U_r h_{t-1} + b_r),\\
u_t &= \sigma(W_u x_t + U_x h_{t-1} + b_u),\\
c_t &= tanh(W_c x_t + U_c(r \odot h_{t-1}) + b_c),\\
h_t &= u_t \odot c_t + (1 - u_t) \odot h_{t-1}
\end{aligned}
$$

wobei die Verknüpfungen Zwischenergebnisse r_t und u_t das *Reset-* und *Updategate* realisieren. σ ist die Sigmoid-Funktion als Aktivierungsfunktion und $\odot$ eine Elementweise Multiplikation. Eine Visualisierung dieser Verknüpfung kann ebenfalls in Abbildung 1 rechts betrachtet werden.

3.3 Long short-term Memory (LSTM

LSTM-Netze nutzen im gegensatz zu den anderen, zwei interne Zustände, h_t und zusätzlich c_t, welche als Kurz- und Langzeitgedächtnis interpretiert werden können. Für die Berechnung neuer Zustände werden drei *Gates* verwendet:

1. **Forgetgate** um gegebenenfalls veraltete, irrelevante Informationen aus dem Langzeitgedächtnis c_t zu entfernen.

2. **Inputgate** um neue Informationen selektiv in das Langzeitgedächtnis c_t zu überführen.

3. **Outputgate** als Filter für die Ausgabe und Zustand h_t.

Die Berechnung der internen Zustände,

$$
\begin{aligned}
f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f),\\
u_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i),\\
o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o),\\
c'_t &= tanh(W_c x_t + U_c(h_{t-1}) + b_C),\\
c_t &= f_t \odot c_{t-1} + i_t \odot c'_t,\\
h_t &= o_t \odot tanh(c_t),
\end{aligned}
$$

implementiert diese filter Mechanismen durch Verknüpfung der Zwischenergebnisse f_t, i_t, o_t. Eine Visualisierung dieser Verknüpfung kann ebenfalls in Abbildung 1 in der Mitte, betrachtet werden.

4 Implementierung

In diesem Kapitel wird die Implementierung und die dabei umgesetzte Parallelisierung vorgestellt.

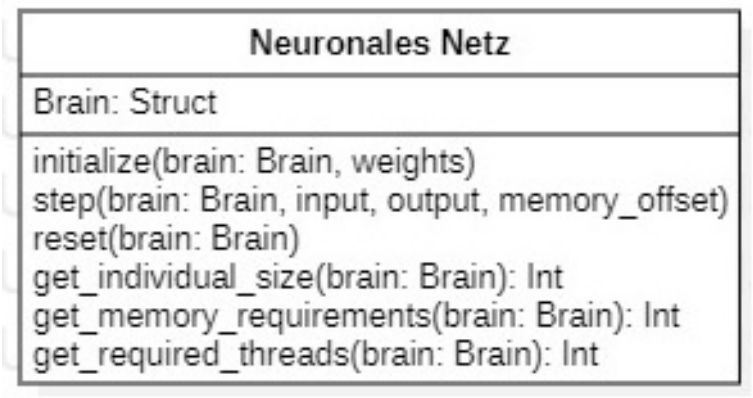

Abbildung 2: Schematisches Klassendiagramm der Neuronalen Netze

4.1 Neuronale Netze

Die Neuronalen Netze wurden durch jeweils eine eigene Klasse wie in Abbildung 2 realisiert. Diese Struktur ist für alle der drei Varianten identisch. Die Gewichtsmatrizen und internen Zustände sowie Anzahl der Eingaben, Ausgaben und Neuronen werden im Struct *Brain* auf dem globalen Speicher der GPU persistiert. Die Methoden und deren Funktion werden im folgenden vorgestellt.

- **initialize(brain, weights): void**
  - Überträgt die Gewichte *weights* in die Gewichtsmatrizen des Netzes.
  - Ruft die *reset()* funktion auf.
- **step(brain, input, output, memory_offset): void**
  - Berechnet die Ausgabe der verstecken Schicht mithilfe der Eingabe *input* wie in Kapitel 3 definiert und abhängig vom RNN-Typ.
  - Berechnet die Ausgabe mithilfe einer Ausgabeschicht und schreibt das Ergebnis in *output*.
  - *memory_offset* ist Offset für den statischen Speicher.
- **reset(brain): void**
  - Setzt die internen Zustände zurück.
- **get_individual_size(brain): Int**
  - Gibt die Anzahl an nötigen Gewichtsparameter des Netzes zurück.
  - Für Optimierung aufgerufen.
- **get_memory_requirements(brain): Int**
  - Gibt die benötigte größe des verwendeten geteilten Speichers zurück.
  - Wird vor GPU-Ausführung aufgerufen.

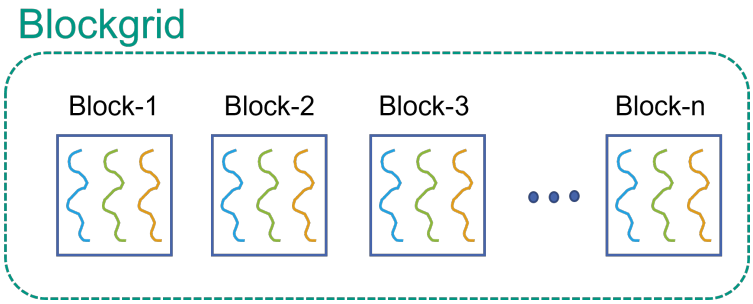

Abbildung 3: Schema eines CUDA-Blocknetz

- **get_required_threads(brain): Int**
  - Gibt die benötigte Anzahl an Threads für die Berechnungen zurück.
  - Wird vor GPU-Ausführung aufgerufen.

4.2 Paralliesierung

Die GPU-Ausführung erfolgt durch den Aufruf einer CUDA-Kernelfunction. Diese wrid auf der GPU als Blocknetz ausgeführt, wobei ein Block aus bis zu 1024 Threads besteht und jeder Block parallel ausgeführt werden kann [3]. In Abbildung 3 ist eine schematische Darstellung eines solchen Blocknetzes zu sehen. Die Anzahl an Blocks und deren Threads, sowie die Größe des benötigten geteilten GPU-Speicher, muss beim Aufruf der Kernelfunktion definiert werden.
Im Trainingsprozess wird diese Kernelfunktion vor jeder Generation ausgeführt woraufhin die Berechnungen der Inidivuuen und der Trainingsumgebungen sowie deren Kommunikation, gänzlich auf der GPU-Ausgeführt werden. Die Parallelisierung ist so implementiert, dass jedem Individuum in einer Generationen ein eigener Block zugewiesen wird, welcher dann unabhängig der anderen Individuen die Evaluation in der Trainingsumgebung übernimmt. Die definierte Anzahl der Blocks muss daher mit der Größe der jeweiligen Population übereinstimmen. Diese Interaktion der beiden Komponenten im Rahmen der Blockparallelisierung ist in Abbildung 4 mit einer beispielhaften Labyrinth Trainingsumgebung illustriert.

Innerhalb einer individuellen Evaluation sind weitere Parallelisierungen durch die Threads implementiert. Jedem Neuron wird ein eigener Thread zugewiesen der dessen Berechnungen durchführt. Die benötigte Anzahl an Threads pro Generation wird durch eine `get-required-threads()`-Funktion bestimmt welche sowohl die Umgebung als auch die Neuronalen Netze implementieren. Das Maximum aus diesen beiden Werten bestimmt dann die Anzahl der benötigten Threads.

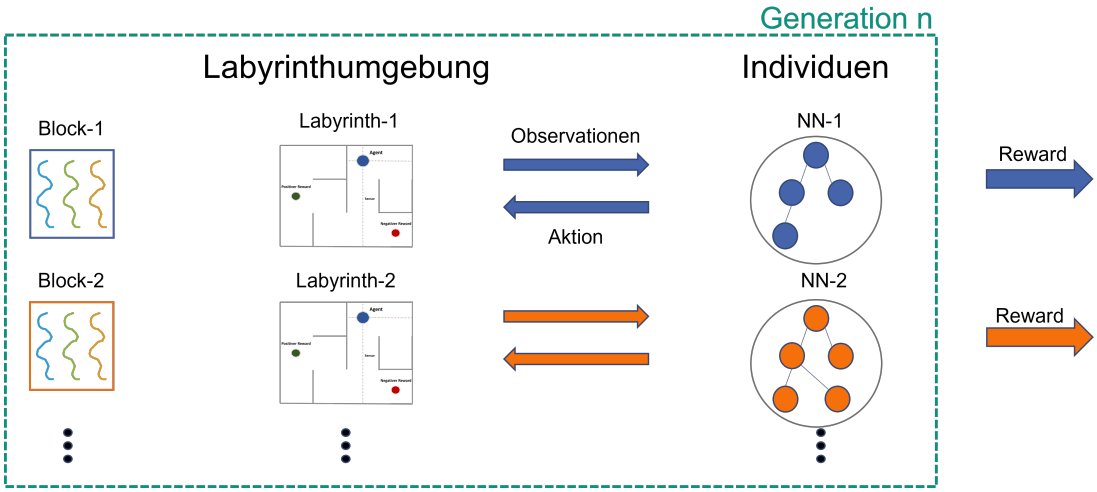

Abbildung 4: Interaktion von Umgebung und Individuen unter Parallelisierung

5 Validierung

5.1 Tests

Für jedes der drei Netze wurde ein eigenes Testskript implementiert das sowohl die richtige Initialisierung als auch die korrekte Ausgabe über beliebige Zeithorizonte überprüft. Die Korrektheit wird Anhand der Machine Learning Bibliothek *Flux.jl* als Referenz getestet. Zur Überprüfung dass die Varianten der RNNs mit denen von *Flux.jl* übereinstimmen wurden diese zunächst auf der CPU implementiert. Die Neuronalen Netze werden sowohl gegen *Flux.jl* als auch gegen die CPU-Implementierung getestet.
Die Unit-Tests umfassen:

- Überprüfung der initialisierten Gewichtsmatrizen und interne Zustände durch die *initialize()* methode.
- Überprüfung der step() funktion
  - Korrekte Ausgabe der Neuronalen Netze über mehrere Zeitschritte.
  - Korrekte Berechnung der internen Zustände über mehrere Zeitschritte.

Die Anzahl der Individuuen die parallel getestet werden sollen, sowie die Anzahl an Zeitschritten, Eingaben und Ausgaben sind dabei konfigurierbar.

5.2 Programmstart

Dieser Abschnitt dient als Anleitung um die Testskripts der Modell auszuführen.

- Zunächst müssen notwendige Pakete geladen werden
  - Über das Terminal in den Projektordner navigieren
  - **`julia -project=.`** ausführen um in die Julia REPL der Projektumgebung zu kommen

  - In den Paketmodus durch Eingeben von `]`
  - **`instantiate`** ausführen um die in der Project.toml Datei definierten Pakete zu laden

- Folgende Skripte können dann über **`julia -project=.`** vom Terminal aus ausgeführt werden
  - **`\test\elman_network.jl`** um die Unittests der Elman-Netze auszuführen.
  - **`\test\gated_recurrent_unit.jl`** um die Unittests der GRUs auszuführen.
  - **`\test\long_short_term_memory.jl`** um die Unittests der LSTMs auszuführen.

6 Fazit

Diese Implementierung Rekurrenter Neuronaler Netze in Julia, mit der umfangreichen Parallelisierung auf der GPU ermöglicht es, die Ausgabe mehrerer hundert Individuen über Tausende Zeitschritte auf herkömmlicher Hardware in annehmbarer Zeit zu berechnen. Diese kurzen Laufzeiten resultieren vor allem durch den hohen Parallelisierungsgrad, aber auch durch effiziente Speichernutzung

Für weitere Arbeiten wäre die Anbindung dieser Netze an das Trainingsskript, sowie die Implementierung weiterer Umgebungen in das NeuroEvolution-Framework denkbar.

Literatur

[1] Author dprogrammer. *RNN, LSTM amp; Gru*. Juni 2020. URL: `http://dprogrammer.org/rnn-lstm-gru`.

[2] Edgar Galván und Peter Mooney. „Neuroevolution in Deep Neural Networks: Current Trends and Future Challenges“. In: *CoRR* abs/2006.05415 (2020). arXiv: `2006.05415`. URL: `https://arxiv.org/abs/2006.05415`.

[3] Nvidia. *Cuda Refresher: The Cuda Programming Model*. Aug. 2020. URL: `https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/`.