Karlsruher Institut für Technologie

Praktikum

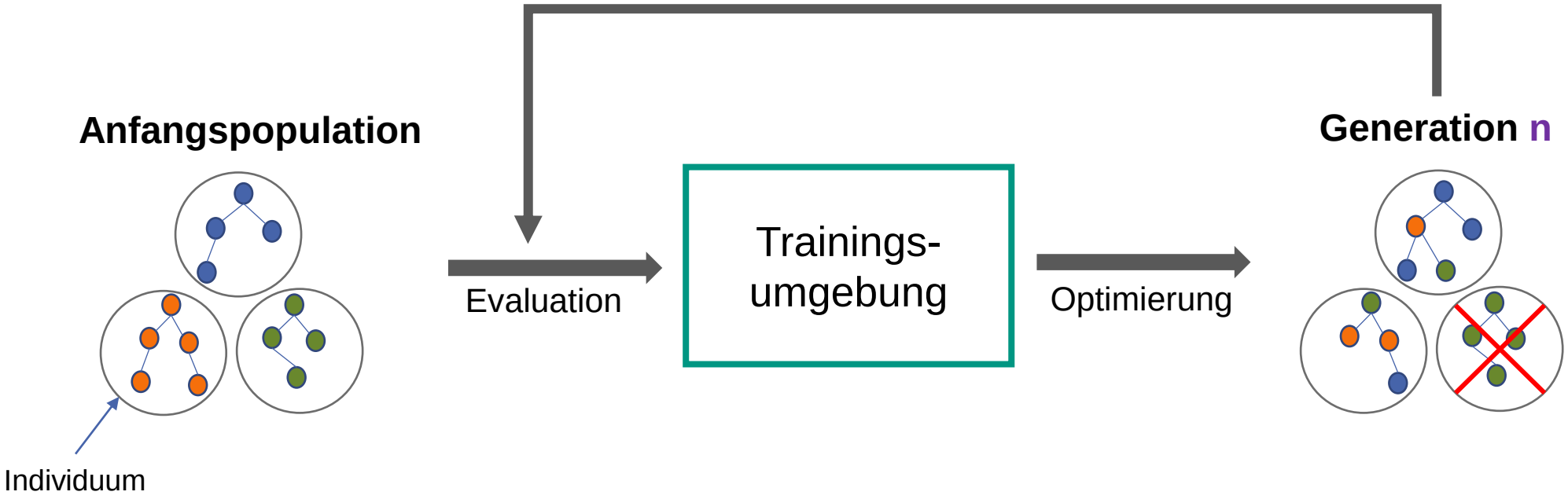
Implementierung tiefer Neuronaler Netze auf der GPU

Julien Aziz
Betreuer: Daniel Zimmermann

Motivation - Neuroevolution

- Methode um Neuronale Netze zu trainieren/generieren
- Population von Neuronalen Netzen
- Evolutionäre Optimierung
  - Mutation, Kombination, Ersetzung
- Evaluation in Trainingsumgebungen

Motivation - Neuroevolution

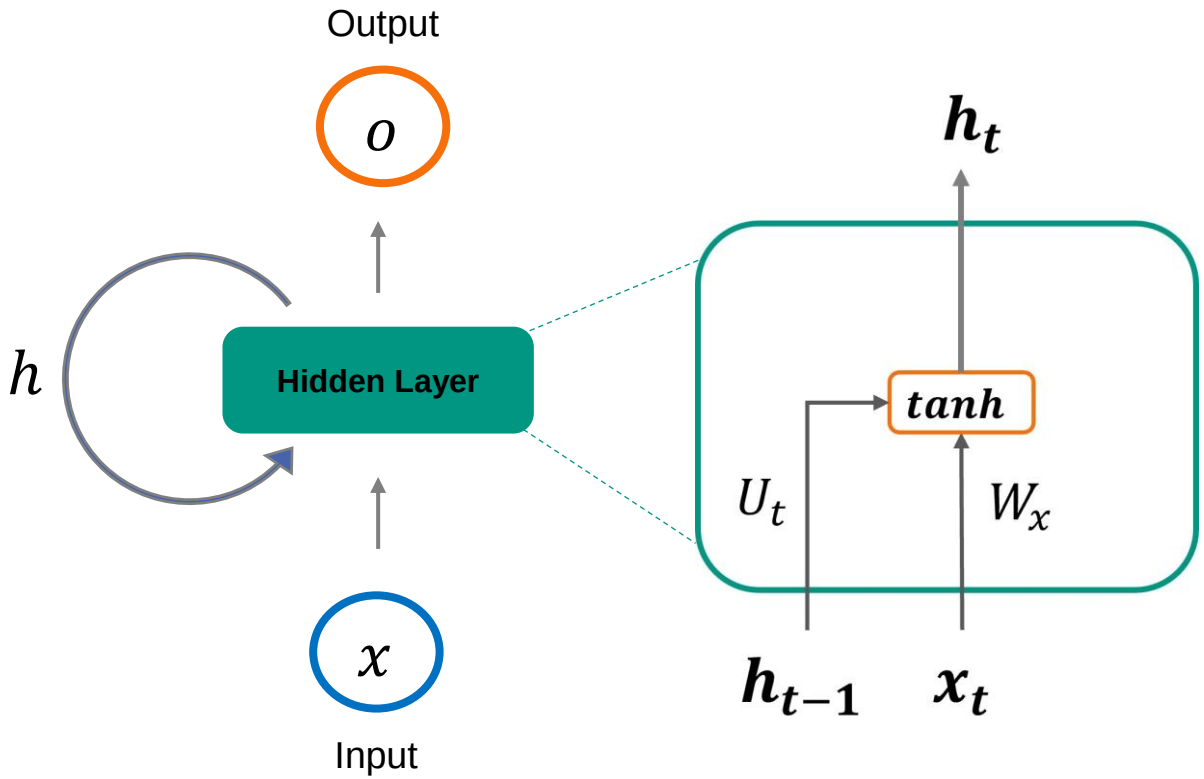


Motivation - Ziel

- Implementierung Neuronaler Netze in Julia
- Ausführung der Hauptfunktionalitäten auf der GPU
- Validierung mit Flux
- Neuronale Netze
  1. Elman-Netz
  2. Gated Reccurent Unit (GRU)
  3. Long Short-Term Memory (LSTM)

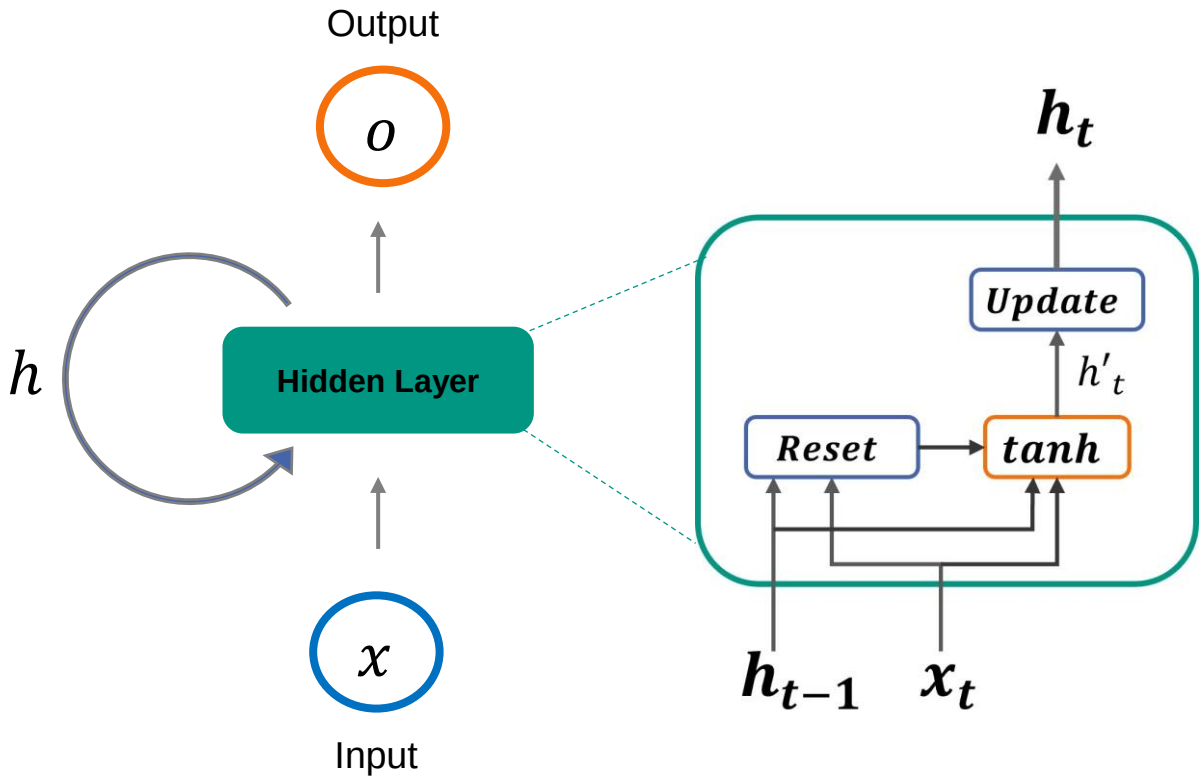
Neuronale Netze - Rekurrenz

- **Elman-Netz**
  - $\mathrm{h_t} = \tanh(\mathrm{W_h x_t} + U_h h_{t-1} + b)$



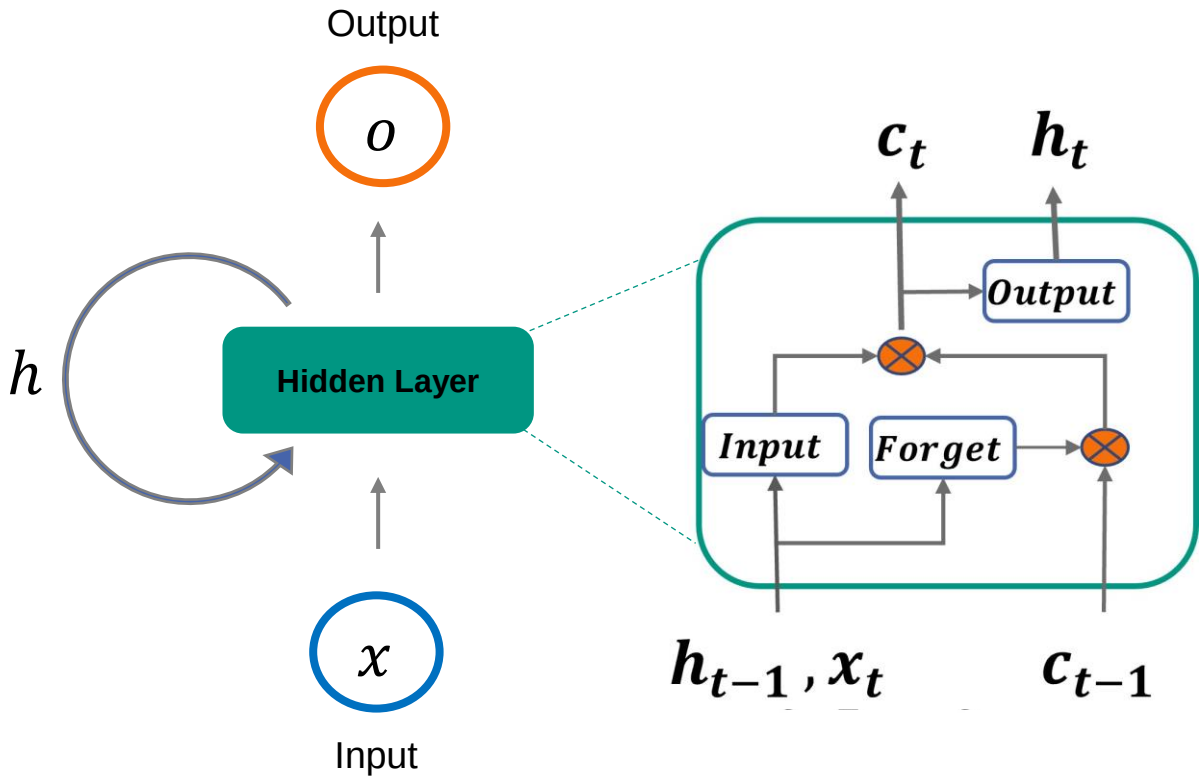
Neuronale Netze - Rekurrenz

- **Elman-Netz**
  - $\mathrm{h_t} = \tanh(\mathrm{W_h x_t} + U_h h_{t-1} + b)$
- **Gated recurrent unit**
  - Update Gate
  - Reset Gate



Neuronale Netze - Rekurrenz

- **Elman-Netz**
  - $\mathrm{h_t} = \tanh(\mathrm{W_h x_t} + U_h h_{t-1} + b)$
- **Gated recurrent unit**
  - Update Gate
  - Reset Gate
- **Long short-term memory**
  - Input Gate
  - Forget Gate
  - Output Gate



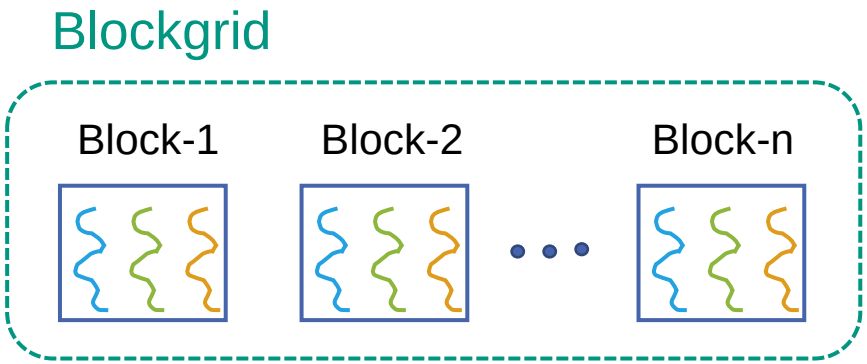
Implementierung - Hauptfunktionalitäten

- NN-Parameter in Struct gespeichert
  - Gewichtsmatrizen
  - Interne Zustände
- **Initialisierung** – Einmalig pro Generation
  - Eingabe: NN-Gewichte von Optimierer
  - Initialisierung des Modells
- **Schrittfunktion** – N-Mal pro Generation
  - Eingabe: Observationen aus Trainingsumgebung
  - Ausgabe: Nächste Aktion des Agenten

Neuronales Netz
Brain: Struct
initialize(brain: Brain, weights) step(brain: Brain, input, output, memory_offset) reset(brain: Brain) get_individual_size(brain: Brain): Int get_memory_requirements(brain: Brain): Int get_required_threads(brain: Brain): Int

Implementierung – GPU Ausführung

- Cuda Kernel Funktion als Einstiegspunkt
  - Ausführung als Blockgrid
  - Jeder Block hat bis zu 1024 Threads
- GPU-Speicher
  - Globaler Speicher
  - Statischer geteilter Speicher
- Vorgehen: Ein Block pro Individuum
  - Ein Thread pro Neuron




```
@cuda threads = x blocks = y shmem = z kernel_function()
```

Implementierung – Parallelisierung

Generation n

Individuen

Trainingsumgebung

Block-1

NN-1

Observationen

Aktion

Labyrinth-1

Agent

Positiver Reward

Sensor

Negativer Reward

Reward

Block-2

NN-2

Labyrinth-2

Agent

Positiver Reward

Sensor

Negativer Reward

Reward

Implementierung – Beschleunigungen

- Überwachung der GPU-Aktivitäten mit “nvprof”
  - Hier: Profiling des GRU Testskripts

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	40.81%	1.11623s	200000	5.5810us	2.1760us	78.271us	julia_getindex_kernel_6568(CuKernelContext,
	32.82%	897.63ms	300011	2.9920us	1.0870us	199.10us	[CUDA memcpy DtoH]
	20.72%	566.85ms	100000	5.6680us	1.9520us	124.83us	julia_getindex_kernel_6263(CuKernelContext,
	5.26%	144.00ms	1000	144.00us	16.799us	2.6583ms	julia_kernel_test_brain_step_5262(CuDeviceAr
	0.37%	10.218ms	2001	5.1060us	1.9840us	320.51us	[CUDA memcpy HtoD]
	0.01%	329.92us	1	329.92us	329.92us	329.92us	julia_kernel_test_brain_initialize_2977(CuDe

- Weitere Beschleunigungen
  - NN-Berechnungen auf statischem GPU Speicher
    - Speicherung der Zwischenergebnisse
  - @Inbounds Annotationen

Implementierung – Validierung

- Zunächst Implementierung auf der CPU
- Flux als Referenz
  - Machine Learning Bibliothek für Julia
- Vergleich der GPU-Implementierung mit CPU & Flux
- Unit Tests der GPU Implementierungen
  - Initialisierung der Gewichte & Zustände
  - Korrekte Ausgabe der Netze über mehrere Zeitschritte

Fazit & Ausblick

- Performante Implementierung Neuronaler Netze auf der GPU
  - Korrekte Ausgaben über lange Zeithorizonte
  - Geringe Laufzeiten mit Standard Hardware
- Laufzeitbeschleunigung
  - Hoher Parallelisierungsgrad
  - Effiziente Speichernutzung
- Implementierung neuer Trainingsumgebungen
  - GUI-Testing

Appendix - Rekurrente Netze

- **Gated recurrent unit**
  - Update Gate: $\mathrm{u_t} = \sigma(\mathrm{W_{ux}x_t} + \mathrm{U_{uh}}h_{t-1} + b_u)$
  - Reset Gate: $\mathrm{r_t} = \sigma(\mathrm{W_{ux}x_t} + \mathrm{U_{uh}}h_{t-1} + b_r)$
  - Memory state: $h'_t = tanh(\mathrm{W_{hh}x_t} + \mathrm{U_{hh}}(h_{t-1} \odot r_t) + b_h)$
  - Output: $h'_t = \mathrm{u_t} \odot h'_t + (1 - u_t) \odot h_{t-1}$
- **Long short-term memory**
  - Input Gate: $\mathrm{i_t} = \sigma\left(\mathrm{W_{ix}x_t} + \mathrm{U_{ih}}h_{t-1} + b_y\right)$
  - Forget Gate $\mathrm{f_t} = \sigma\left(\mathrm{W_{fx}x_t} + \mathrm{U_{fh}}h_{t-1} + b_f\right)$
  - Output Gate $o_\mathrm{t} = \sigma(\mathrm{W_{ox}x_t} + \mathrm{U_{oh}}h_{t-1} + b_o)$
  - Memory state: $c'_\mathrm{t} = \sigma(\mathrm{W_{cx}x_t} + \mathrm{U}_{cx}h_{t-1} + b_c)$
  - Long-term: $c_\mathrm{t} = f_t \odot c_{t-1} + i_t \odot c'_t$
  - Output: $o_t \odot \sigma(c_t)$

Appendix Trainingsframework