NeuroEvolution of Augmenting Topologies (NEAT)

Steuerung eins Mario Jump & Run Spiels

Jan Urfei
Bonn-Rhein-Sieg University of Applied Sciences
Grantham-Allee 20
53757 Sankt Augustin, Germany
jan.urfei@inf.h-brs.de

Tim Lügger
Bonn-Rhein-Sieg University of Applied Sciences
Grantham-Allee 20
53757 Sankt Augustin, Germany
tim.luegger@inf.h-brs.de

ABSTRACT

Ziel war es eine Mario Spielfigur mit bestimmten Szenarien zu trainieren, sodass diese dann das/die Level best möglichst lösen kann.

1. GENERAL NOTES

- Do not state things you cannot confirm either by literature or experiment. This is science, not black magic.
- Normalize your data and resulting statistical descriptors (like RMSE, μ)
- Save your experimental results to disk **before you visualize**. You **will** change your plots quite a lot after gathering the data.
- Don't submit reports with many pages containing just a single figure. Scrolling is terrible for our health.
- **Separate** training and test data. Don't use test data in your algorithm to learn or make decisions, they are only allowed to test the end result. If you need a separate sample set to make decisions during training or optimization, create a third (validation) set. Only your test data will really tell you how good your algorithm is.

1.1 Algorithm Parametrization

Develop an actual strategy, preferably on a reduced but similar problem. You can reduce the number of samples, the targetted number of time steps your controller runs in a simulation, or any other non-destructive problem reduction method. If your algorithm has two paramers you need to adjust, it should be no problem to take 5 *sensible* values per parameter and compare all combinations. For stochastic algorithms, like evolutionary approaches, make sure you **repeat your experiments** at least 5-10 times, depending on the amount of randomness. Since your algorithm makes "random" changes, just comparing single runs does not give you a good estimation on its performance. Do **not** pick your values such that they only confirm the values you **want** to use.

1.2 Structure

1. **Assignment Description**: first provide a brief description of the assignment that was handed out to you. This includes a description of the data.
2. **Approach**: describe the algorithm
3. **Experiments**: describe your experimental setup, algorithm parameters, data preprocessing first. Then include the results and a discussion thereof.
4. **Conclusion**: any conclusions about the algorithm, bugs, future work.

1.3 Visualization

- Label your axes
- Use logarithmic scale when appropriate
- Use descriptive captions below your figures
- Add legends if necessary
- Make fontsizes large enough, linewidths thick enough to be readible in the final report.
- When making comparisons, make sure the results are either in the **same** graph or graphs are plotted next to (or close to) each other.
- **In short: make sure people can read your figures**

2. ASSIGNMENT

Aufgabe war es den für die HeartRate Prediction implementierten NEAT Algorithmus, der sich an dem Originalpaper [1] orientiert, so zu verändern und anzupassen, dass er ein Problem eines Projektes löst, das sich ausgesucht werden konnte.

In unserem Fall hieSS das, eine Mario Figur durch ein Level zu bringen. Dafür muss eine neue Parametrisierung gefunden werden.

Als Trainingsdaten konnten Level generiert werden, die einen gewünschten Schwierigkeitsgrad besaSSen. Die Daten, die man aus dieser Map ziehen konnte, sind die Blöcke zentriert um den Mario herum (siehe Figure 1). Dabei ist jedem Blocktyp eine eigene Id zugewiesen. Aufgrund von diesen Daten soll NEAT eine Tastenkombination wählen, um den Mario zu steuern.

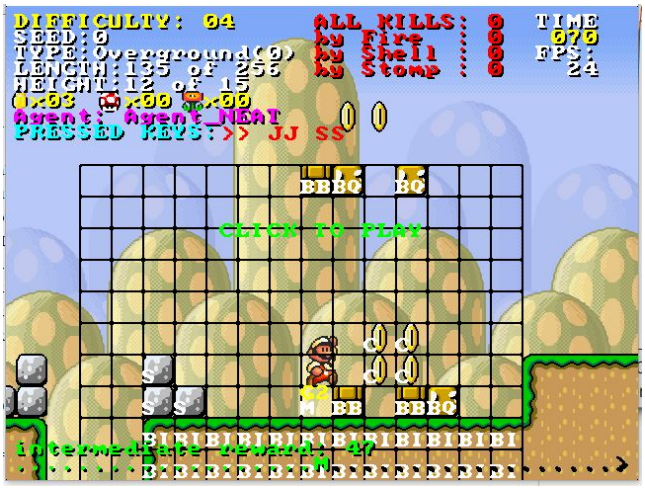

Figure 1: Umgebungsblöcke des Marios für den Input

3. APPROACH

Das Herzstück von NEAT ist nahe des Papers implementiert worden. Der rudimentäre Ablauf ist auch in dem folgenden Pseudocode zu betrachten.

Algorithm 1 NEAT

```
1: function TRAIN(Parameters)
2:     Parameter: ← defineSpecies(Parameter)
3:     sort(fitness)
4:     Parameter: ← sharedFitness(Parameter)
5:     Parameter: ← defineBigSpecies(Parameter)
6:     elitismInBigSpecies(Parameter)
7:     Parameter: ← defineOffspringFromSpecies(Parameter)
8:     for species s in S)
9:         Parameter: ← crossover(s,randomSpecies)
10:        Parameter: ← mutate(s)
       /* mutateWeihts,add Node/Connection or Disable
   Connection*/
11:        Parameter: ← appendRandomGenomes()
       /* Until the population is full*/
12:    end for
13: end function
```

Im Gegensatz zum Original Paper habe wir noch implementiert, dass Kanten wieder Disabled werden können, damit die Topologien minimal bleiben können.

Eine besondere Herausforderung bestand darin, die Kommunikation zwischen dem Simulator (der in Java implementiert war) und unserem NEAT Algorithmus (der in Mathlab implementiert war) herzustellen. Dazu haben wir uns einer Bibliothek bedient, mit Hilfe deren man aus Java heraus Variablen in den Matlab Workspace schreiben kann. Der Simulator holt sich dann die verschiedenen Netze aus dem Workspace und berechnet an Hand des aktuellen Spielfeldes den Output bzw. den daraus resultierenden Tastendruck. Um die Kommunikation zwischen Matlab und Java möglichst gering zu halten (da diese immer sehr viel Zeit gekostet hat) ist die Berechnung der Gewichtsmatrixen, als auch der Output in Java implementiert. Hat der Simulator das Spiel soweit gebracht bis es entweder gewonnen ist, die Zeit abgelaufen ist oder der Mario gestorben ist bestimmt er eine Fitness für jede Topologie. Diese wird zurück in den Matlabworkspace geschrieben und das oben angedeutete Training ausgeführt. Danach beginnt der Prozess von neuem. In der folgenden Abbildung (Figure 2) ist der Ablauf nochmals veranschaulicht.

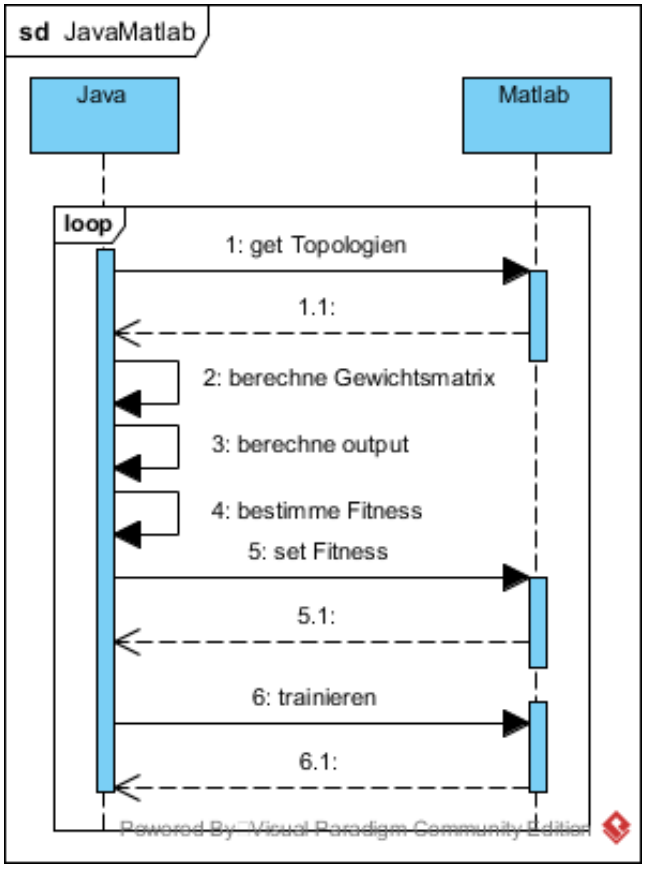

Figure 2: Kommunikation und Ablauf zwischen Java und Matlab

Um die Trainingszeit noch schneller zu machen, haben wir ein kleineres Sichtfeld um den Mario genommen, welches lediglich 10 x 10 groSS ist. Somit werden die Netze wesentlich kleiner. Des Weiteren haben wir das Sichtfeld auf den Teil rechts vom Mario beschränkt, da in unseren Level keine

Rückwärtsbewegung notwendig war.

Zusätzlich um das lernen der Blöcke zu erleichtern, haben wir den Blockids einen Verlauf gegeben. So haben ähnliche Blöcke, wie z.B. welche, die Hindernisse darstellen, ähnliche Blockids. So haben Blöcke die von groSSer Bedeutung für das erreichen des Ziels sind eine groSSe Blockid (z.B. der Boden) und Blöcke die nicht so wichtig sind eine niedrige (wie z.B. Münzen).

4. EXPERIMENTS

Experiments are repeated 10 times. By the way, this is not enough for a description.

4.1 Parameterization

This would include a description of all parameter tuples and a description on how you reduced the problem to provide small and fast runs for this extensive comparison. Results would be shown in boxplots, as these provide you with the median and 25% and 75% percentile for every parameter tuple. This makes comparison easy, as you can plot multiple boxplots next to each other, similar as in Figure 6.

4.2 NEAT vs ESP

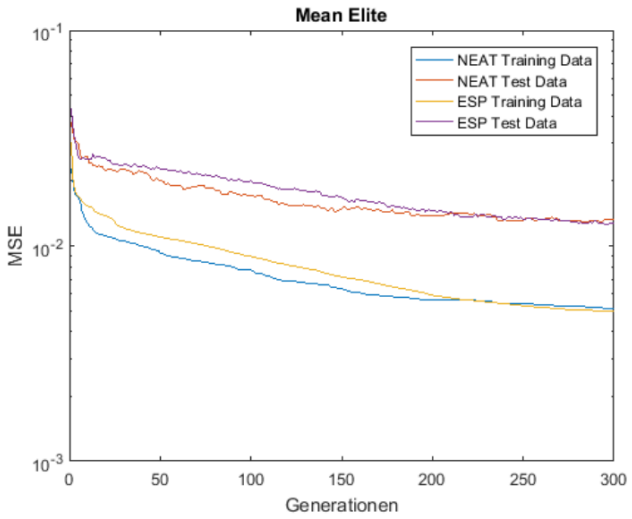

Figure 3: Development of average mean square error, comparing NEAT and ESP

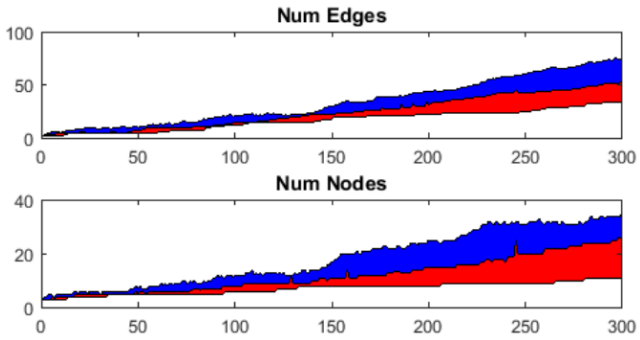

Figure 4: Development of min/mean/max number of nodes and edges. *Legend is missing*!

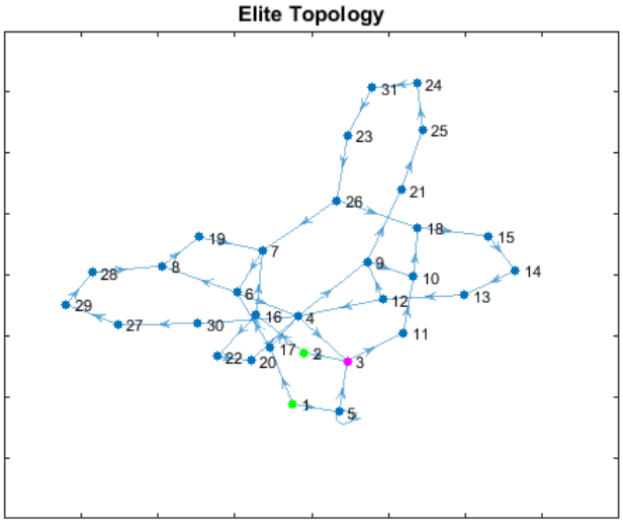

Figure 5: Elite topology. Nodes are assigned their IDs. Green are input nodes, purple are output nodes.

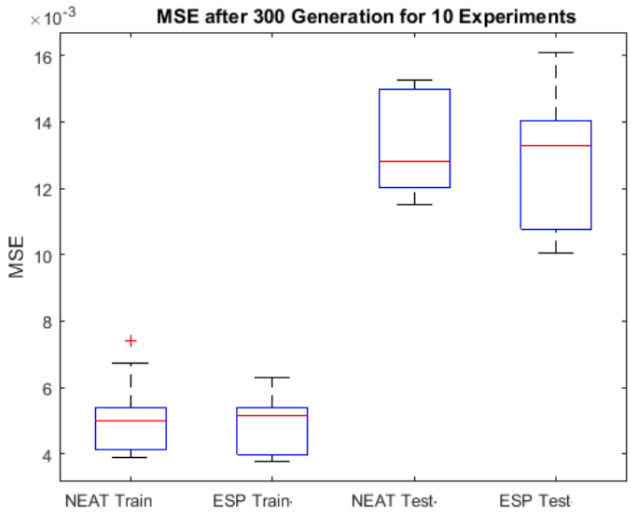

Figure 6: Comparing MSE of NEAT and ESP

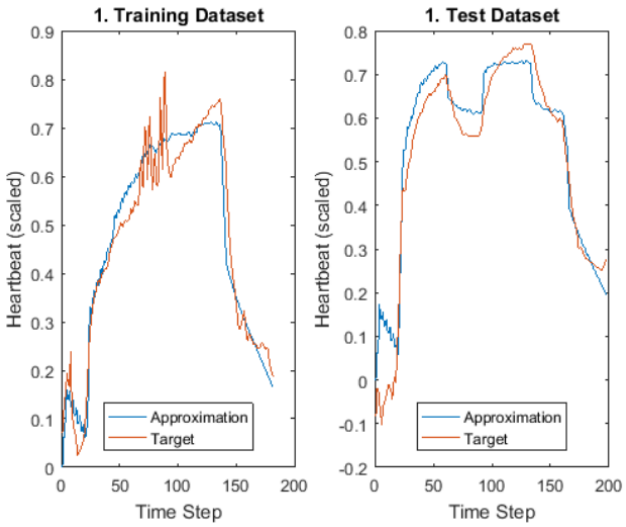

Figure 7: Approximated heart rates after 300 generations

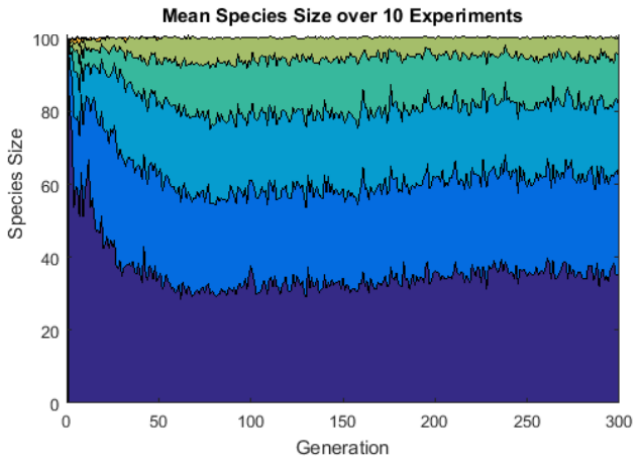

Figure 8: Development of species over time, averaged over 10 generations

5. CONCLUSION

6. REFERENCES

[1] K. O. Stanley and R. Miikkulainen. Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation*, 10(2):99–127, 2002.