Finding new bio-plausible Learning Rules using Deep Reinforcement Learning

Jamie Lohoff[1,2] and Emre Neftci[1,2]

[1] *Peter Grünberg Institut 15, Forschungszentrum Jülich GmbH, Jülich, 52425, Germany,*

[2] *Fakultät für Elektro- und Informationstechnik, RWTH Aachen, Aachen, 52074, Germany*

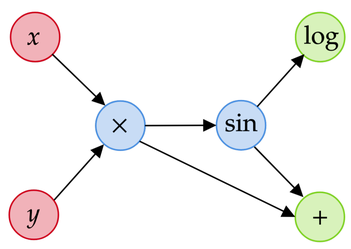
Abstract

Gradient-based learning is still the best bet when training spiking neural networks on supervised tasks. Although backpropagation, the state-of-the-art in modern AI, is not bio-plausible, there exists a wide range of approximations with this property that achieve competitive performance, e.g. e-prop[1,2]. We propose a new framework called **AlphaGrad** that could find more such learning rules by systematically exploring the search space using Deep Reinforcement Learning and methods from Automatic Differentiation(AD).

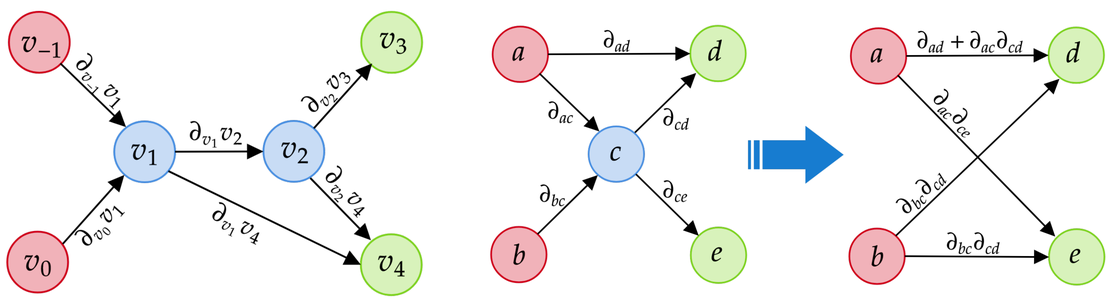
Premise: Everything is a computational graph!

Neuron models, neural networks, ODEs and many other things in quantitative modeling can be expressed as **computational graphs**.
These are directed acyclic graphs where every vertex corresponds to an elemental mathematical operation such as an addition or a logarithm.
The edges describe the data dependencies. Below is an example of a computational graph.

$$f(x,\ y)\ =\ \begin{pmatrix} \log\sin(xy) \\ xy\ +\ \sin(xy) \end{pmatrix}$$



Often, we need the gradient or Jacobian of such a graph to perform some optimisation task, e.g. training a neural network or applying some solver, e.g. Newton-Raphson. A result from AD states that we can compute the derivative in the graph picture through[3]:



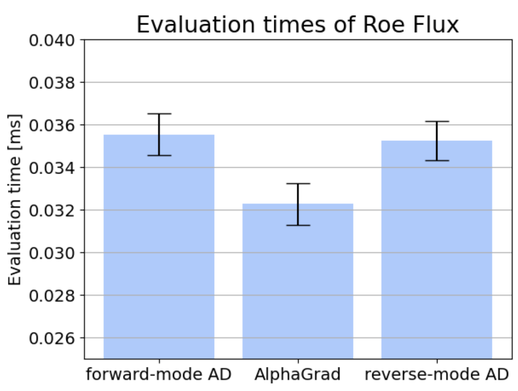
1.) We start by assigning the derivative of the posterior vertex w.r.t. the prior vertex to the respective edges.

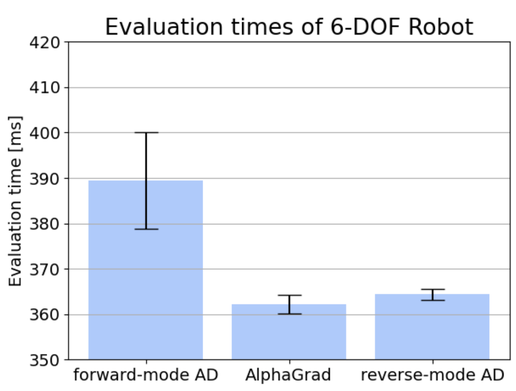
2.) We use the **vertex elimination rule** to remove vertices from the graph and accumulate the derivatives.

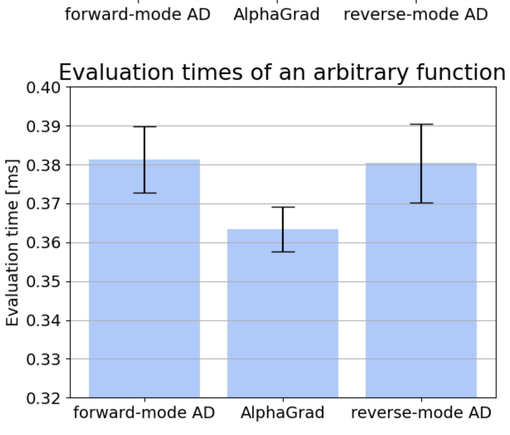
If we eliminate all the interior vertices, the remaining edges contain the **full Jacobian** of the computational graph. The **elimination order** has an serious impact on the computational cost. Can we find the optimal order w.r.t. number of multiplications, memory usage or walltime? Is it possible to use this framework to compute approximations of Jacobians?

Results: So...was this good for anything?

The following results are unpublished, cherry-picked and not yet peer-reviewed.
We started by using the **number of multiplications** as our optimization target for AlphaGrad and measuring the **actual speed** up we would gain from that:







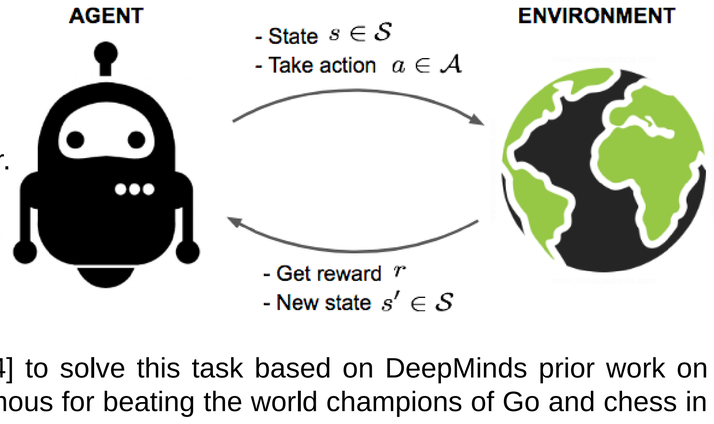
Sadly, we were not able to create a speedup for neural networks, as backpropagation is mathematically speaking the best you can do for scalar functions in terms of number of multiplications. For **memory consumption** or **pure walltime**, things will look different though.

All experiments were done on CPU.

Method: Making a chess AI do cool things

But how do we optimize for metrics of the Jacobian accumulation such as the number of multiplications, memory requirements or walltime?
Entry **Reinforcement Learning** (RL). It allows us to optimize for almost every goal. RL is very different from supervised learning. At the heart of RL lies the **environment-interaction loop,** which exemplifies this.

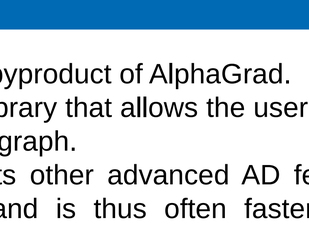
In our case:
- **Agent**: Neural Network and sampling method (here MCTS)
- **Environment**: Comp. Graph.
- **State**: computational graph repr.
- **Action**: eliminate a vertex
- **Reward**: The goal we wish to optimize for, e.g. number of multiplications or walltime



We decided to use **AlphaZero**[4] to solve this task based on DeepMinds prior work on AlphaTensor[5]. This agent is famous for beating the world champions of Go and chess in a landslide victory.

What are its key components?
1.) **Neural Network**

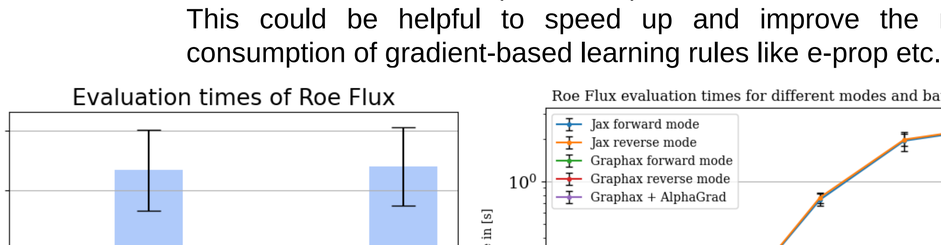
Comp. Graph Repr., e.g. adjacency matrix



Probability distribution over vertices to eliminate next

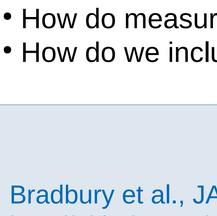
Estimate of the cost/reward of the vertex to eliminate

2.) **Monte-Carlo Tree Search (MCTS)**



MCTS is used to sample actions from observations by extrapolating the impact of the selected action on the overall reward.
It performs a number of MC simulations to estimate the outcome of the entire game.
Our neural network guides the tree search to make it more efficient.
The probability distribution output is used to select an action in the expansion phase.
The reward estimation is used to replace the simulation/rollout phase.

Outlook: Where is the Neuromorphic Stuff?



Graphax is a byproduct of AlphaGrad.
It is a JAX[6] library that allows the user to do vertex elimination on a computational graph.
It also supports other advanced AD features such as row/column compression and is thus often faster than corresponding JAX function transformations (see below).
This could be helpful to speed up and improve the memory consumption of gradient-based learning rules like e-prop etc.

The **grand goal** is to augument the vertex elimination method in such a way that it allows *systematic search for gradient-based, bio-inspired learning rules*.

Unanswered questions are:
- How do we do the approximation in a mathematically meaningful way?
- How do measure the quality of the approximation?
- How do we include the temporal components?

References

[1] Bellec et al.: A solution to the learning dilemma for recurrent networks of spiking neurons, 2020, Nature Communications

[2] Zenke&Neftci: Brain-inspired Learning on Neuromorphic Substrates, 2020, Proceedings of the IEEE

[3] Griewank&Walther: Evaluating Derivatives, 2008, SIAM

[4] Silver et al.: Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm, 2017, Nature

[5] Fawzi et al.: Discovering faster matrix multiplication algorithms with reinforcement learning, 2022, Nature

[6] Bradbury et al., JAX: Composable transformations of Python+Numpy programs, http://github.com/google/jax