MCFC findings

Bhumika Mistry

March 22, 2023

Abstract

This report contains a series of notes on my findings on using Mass Conserving Neural Networks for learning basic arithmetic - specifically addition and multiplication using Mass Conserving Fully Connected (MCFC) networks. The models are evaluated on a benchmark specifically designed to test the extrapolation abilities of arithmetic networks. On the two input tasks, MCFC networks can perform with full success over the tested ranges. Code is found in a branch named 'hoedt-mcfc': `https://github.com/bmistry4/nalm-benchmark/tree/hoedt-mcfc`.

1 Experiment

I use the Single Layer Arithmetic experiment setup and evaluation from Mistry et al. (2022, Section 6.5). Unless mentioned otherwise, assume the experiment hyperparameters are the same as the original (Mistry et al., 2022, Appendix B, Table 5).

2 Models

2.1 Addition

MCFC: I use the MCFC architecture used in the Static Arithmetic Task from Hoedt et al. (2021, Appendix B.1.3). For these experiments, there is no need for an FC layer like in Hoedt et al. (2021, Appendix B.1.3) and the model size has 2 inputs and 1 output. Except for this, there are no other changes to the architecture.

The *out_gate* uses a sigmoid normaliser and the *junction* uses an softmax normaliser. As there is only 1 output, the softmax normaliser always outputs 1's so the LinearRedistribution's $\mathbf{r}$ parameter can have any value. As the auxiliary vector is just a scalar and the output size is 1, the *out_gate* is simplified to the sigmoid of the summation of the linear layer's weight and bias i.e., $\sigma(w + b)$. Hence, for a model instance to be successful in (2-input) addition requires the follow condition to be met:

$$1 = \sigma(w + b)$$
$$\sigma^{-1}(1) = w + b,$$

which in Pytorch would mean the sum of the parameters would be greater than 16.7 (1 d.p.).[1].

The regularisation is the same as what is used in Hoedt et al. (2021)[2], which is used along with the same piecewise regularisation scaling as the NAU or NMU. Scaling occurs during epochs 20,000 to 35,000 to be consistent with (Mistry et al., 2022).

2.2 Multiplication

MulMCFC: I use the MulMCFC architecture used in the Static Arithmetic Task from Hoedt et al. (2021, Appendix B.1.3). Again, there is no need for an FC layer like in Hoedt et al. (2021, Appendix B.1.3) and the model size has 2 inputs and 1 output. There is no 'trash cell', so no indexing is required[3]. A bias (α) for the MulMCFC is still learnt, but we expect it to converge to a value of zero for this task.

The regularisation is the same as what is used in Hoedt et al. (2021)[4], which is used along with the same piecewise regularisation scaling as the NAU or NMU. Scaling occurs during epochs 20,000 to 35,000 to be consistent with (Mistry et al., 2022).

Note that a limitation of the MulMCFC is its inability to process negative numbers due to the log. To alleviate this, I test two additional modules which modify the MulMCFC to also work with negatives by treating the inputs as only positive numbers and then reapplying the appropriate sign to the module output. In other words, I apply a sign correction mechanism. To do this, I use the sign correction mechanisms of the iNALU and Real NPU - see Mistry et al. (2022, Section 5.1) for an explanation.

MulMCFC (sign: iNALU): This uses the mixed sign vector from Schlör et al. (2020), where I replace the $\mathbf{W}$ with $\mathbf{r}$. No additional parameters require to be learnt.

MulMCFC (sign: Real NPU): The sign retrieval of the Real NPU is defined as $\odot \cos(\mathbf{W}^{\mathrm{RE}}\mathbf{k})$. I replace the $\mathbf{W}^{\mathrm{RE}}$ with $\mathbf{r}$ and $\mathbf{k}$ is calculated the same as how the Real NPU does it. This method requires learning a gating vector which also has a regularisation loss applied to enforce parameters to go to either 0 or 1.

[1] the exact value of $\sigma^{-1}(1)$ is somewhere within the range (16.6,16.7]

[2] `https://github.com/bmistry4/nalm-benchmark/blob/hoedt-mcfc/stable_nalu/layer/mcfc.py#L56`

[3] E.g., see indexing in `https://github.com/hoedt/stable-nalu/blob/master/stable_nalu/layer/mcfc.py#L74`

[4] `https://github.com/bmistry4/nalm-benchmark/blob/hoedt-mcfc/stable_nalu/layer/mcfc.py#L56`

2.3 Subtraction and Division

I did not run these experiments as I was unsure how to create a mass conserving model which could learn extrapolative subtraction/division, without relying on an extra FC layer.

3 Evaluation

I use the same evaluation metrics as Mistry et al. (2022) which measures the success rate, convergence speed and sparsity error.

For calculating the sparsity errors in MCFC networks, I calculate the mean over the following:

- MCFC junction sparsity error: How far way the values are from 1/0 once the softmax is applied. For this task the error will always be 0.
- MCFC out gate sparsity error: How far away the values are from 1/0 once the sigmoid is applied.
- MulMCFC bias sparsity error: How far away the values are from -1/1/0.
- MulMCFC (sign: Real NPU) g sparsity error: How fast the gate values are from 1/0.

4 Results

Confidence intervals are calculated over 25 seeds. To access experiments shell script click here.

4.1 Addition

Figure 1 shows the MCFC can robustly learn addition. The convergence speed is slower than the NAU, but considering that all convergences occur after 20,000 iterations, it implies that the success speed is correlated with when the regularisation switches on. The low sparsity error indicates that the parameters are converging as expected.

4.2 Multiplication

Figure 2 shows the MulMCFC cannot learn any ranges which can contain negative inputs as expected. Both the MulMCFC adaptation which has a sign retrieval mechanism alleviate the issue. Similar to the MCFC with addition, all the MulMCFC variants rely on the regularisation to occur before the model converges to an extrapolative solution and is slower to converge than the NMU. Sparsity errors for all successful models are low indicating that extrapolative solutions were found. The differences in performance between the iNALU and RealNPU versions are minimal.

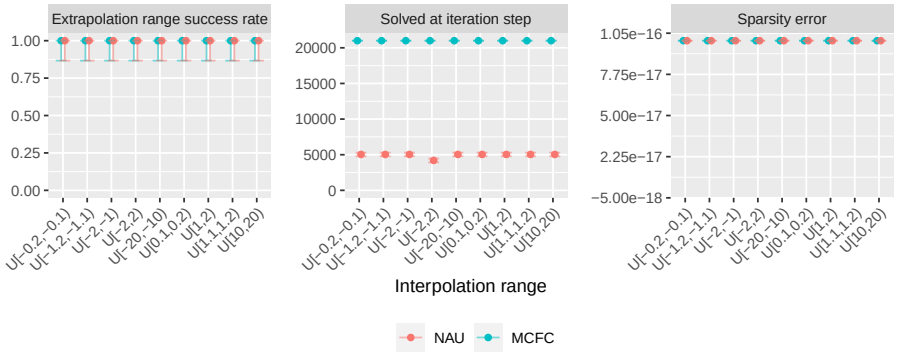

Figure 1: 2-input addition

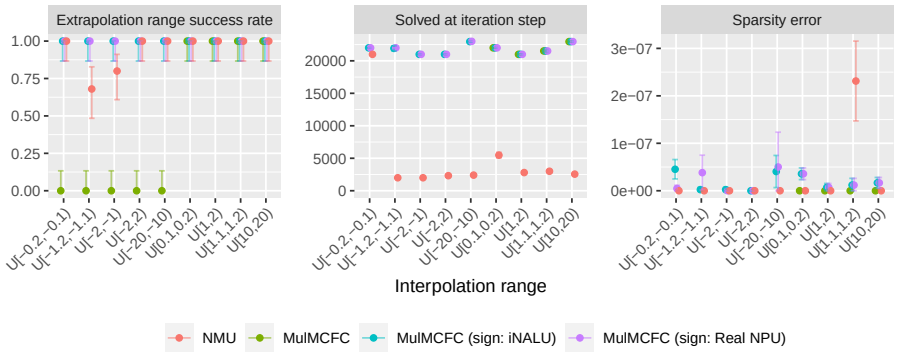

Figure 2: 2-input multiplication

5 Future directions

To extend these findings, the following suggestions could be taken:

- Run experiments with redundancy e.g., 10 inputs where only 2 are relevant.
- Create extensions for a module which can do multiplication and division. Preferably, such modules would not need an additional FC layer but be a single module.
- Test on inputs which require alternate modalities e.g., arithmetic MNIST tasks.
- Combine the different modules to form a multi-operation module which is used in a symbolic regression setting.

References

Pieter-Jan Hoedt, Frederik Kratzert, Daniel Klotz, Christina Halmich, Markus Holzleitner, Grey Nearing, Sepp Hochreiter, and Günter Klambauer. MC-LSTM: mass-conserving LSTM. *CoRR*, abs/2101.05186, 2021. URL `https://arxiv.org/abs/2101.05186`.

Bhumika Mistry, Katayoun Farrahi, and Jonathon Hare. A primer for neural arithmetic logic modules. *Journal of Machine Learning Research (JMLR)*, 23 (185):1–58, 2022. doi:10.48550/ARXIV.2101.09530.

Daniel Schlör, Markus Ring, and Andreas Hotho. iNALU: Improved neural arithmetic logic unit. *Frontiers in Artificial Intelligence*, 3:71, 2020. ISSN 2624-8212. doi:10.3389/frai.2020.00071.