Lightweight Neural Programming: The GRPU

Felipe Carregosa[1(✉)], Aline Paes[2], and Gerson Zaverucha[1]

[1] Department of Systems Engineering and Computer Science, Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ, Brazil
{fborda,gerson}@cos.ufrj.br
[2] Department of Computer Science, Institute of Computing, Universidade Federal Fluminense, Niterói, RJ, Brazil
alinepaes@ic.uff.br

Abstract. Deep Learning techniques have achieved impressive results over the last few years. However, they still have difficulty in producing understandable results that clearly show the embedded logic behind the inductive process. One step in this direction is the recent development of Neural Differentiable Programmers. In this paper, we designed a neural programmer that can be easily integrated into existing deep learning architectures, with similar amount of parameters to a single commonly used Recurrent Neural Network. Tests conducted with the proposal suggest that it has the potential to induce algorithms even without any kind of special optimization, achieving competitive results in problems handled by more complex RNN architectures.

Keywords: Recurrent Neural Networks
Neural Differentiable Programmers

1 Introduction

Recently there has been a renewed interest in merging traditional programming and Neural Networks (NNs), particularly thanks to more advanced Automatic Differentiation (AD) tools [8]. These new tools can evaluate functions written in the host languages idiomatic structures, allowing programmers to easily and efficiently obtain the gradient of varied units of code with respect to their arguments. This enables augmenting the programming toolset with the Machine Learning capabilities.

With a similar goal, Neural Differentiable Programmers (NDPs) [9,11] have been developed to allow NNs to compose algorithms in more traditional ways. This allows them to potentially tackle hard problems, involving complex arithmetic and logical reasoning. Thus, in order to model the input-output relationship, instead of applying a series of transformations directly over the input, NDPs

The authors would like to thank the Brazilian Research Agencies CNPq and CAPES for partially finance this research.

V. Kůrková et al. (Eds.): ICANN 2018, LNCS 11141, pp. 218–227, 2018.
https://doi.org/10.1007/978-3-030-01424-7_22

choose a sequence of transformations from a predefined instruction set, yielding an explicit algorithm to transform the input into the solution. Furthermore, they can also decouple its learned logic from the specific input values, allowing for better generalization and re-usability in different contexts. However, current NDP models focus on end-to-end solutions for specific contexts and problems, instead of being easily integrated into current Deep Learning models.

In this paper, we propose The Gated Recurrent Programmer Unit (GRPU), a NDP technique that can be easily integrated into any current model that uses a Recurrent Neural Network (RNN). Moreover, GRPU uses around the same amount of parameters as a simple Gated Recurrent Unit (GRU) [4], is agnostic in terms of external memory structure and data inputs, and can be extended in similar ways to RNNs, like stacking and soft attention strategies. This way it can provide a lightweight way of augmenting Deep Learning models with the induction of more traditional programs.

The rest of the paper is organized as follows. The next section briefly explains the GRU and the most known NDPs in the literature. The 3rd section details the model devised in this work. The 4th section brings the experiments we have conducted in this work, and the last section is the conclusion.

2 Preliminaries

Here we briefly explain the GRUs, by which our model is inspired, and the most relevant neural programmers found in the related literature.

2.1 Gated Recurrent Unit (GRU)

Recently, a new, simpler, architecture for RNNs has been developed, the *Gated Recurrent Unit* (GRU) [4]. GRUs present comparative performance to the traditionally used *Long Short-Term Memory* (LSTM) [5], while using fewer parameters, as they have only two interacting layers instead of three: the *update gate* and the *reset gate*. When the value computed at the reset gate is close to 0, the corresponding previous hidden state is erased and, therefore, ignored when creating the new state. This allows the GRU to drop information judged irrelevant. The update gate, on the other hand, controls how much information from the previous hidden state should be directly carried over to the current hidden state. This shortcut between the previous state and the following one allows information to be kept untouched indefinitely, helping with the Vanishing Gradient Problem [3].

The value of the current hidden state is computed as $h_t = (1 - u_t) * h_{t-1} + u_t * \tilde{h}_t$, where **u** and **r** stands for the update and reset gates, respectively, and $\tilde{h}_t$, the new candidate state: $\tilde{h}_t = \tanh(W.[r_t * h_{t-1}, x_t] + b)$. The values of the update and reset gates are defined with their own set of parameters, where the update gate is computed as $u_t = \sigma(W_z.[h_{t-1}, x_t] + b_u)$ and the reset gate as $r_t = \sigma(W_r.[h_{t-1}, x_t] + b_r)$.

2.2 Related Work: Neural Programmers

Neural Differentiable Programming (NDP) techniques try to combine the pattern matching and universal approximation nature of the neural networks with the discrete series of operations from traditional algorithms [9]. Fundamentally, neural networks are simply a chain of geometric transformations, and finding one of such transformations that can fully generalize each traditional operation, such as arithmetic and logic operations, is hard and require potentially large amounts of data. For example, even a simple sum or product of numbers is not a trivial task for a neural network to learn, especially considering the distortion caused by the non linear transformations that occur at each step.

Integrating algorithmic-like aspects has been a tendency since the success of the attention models [12]. They allow the network to learn to choose the data it wants to access in a completely differentiable way. NDPs go one step further and not only apply the selection to the input data, but also to the operation applied to the data. For that, they comprise a selection of differentiable operations, and through soft attention they are able to select an operation for each step, and the results of each step can then become the input of the following step. They possess, then, the ability to induce algorithms that transform the original input into the desired output through the multiple steps. The selection operation usually has the form $result = oplist(args)^T softmax(opcode)$, where *oplist* is an N-sized vector in which each field is an operation like sum or multiplication, and *opcode* is a vector with N values, generated by a RNN at each step.

Some of the most notable neural programmers are:

- The *Neural Programmer* [9] is a table query based model that, given an input question, selects a series of aggregate operations and a series of columns from the input table for each operation to be applied. The training phase involves finding the operations and column arguments that minimizes the error towards the given output, using two LSTMs and two softmax layers.
- The Neural Programmer Interpreter [10] is composed of a single LSTM and a domain specific encoder for the state of the environment. The LSTM has three selector units to choose the next operation, its arguments, and when the subprogram terminates. It predicts the next step of a program only, and not the full program at once, requiring the program trace as input.
- The *Neural Random Access Machines* [7] is a sequence-to-sequence programmer model, in which every data register of the virtual machine it implements contains a pointer (a probability distribution) that can be transformed into new pointers through look-up-table based operations. Each pointer can be used to read or write from a memory tape using attention.

3 The Gated Recurrent Programmer Unit

We introduce a novel neural differentiable programmer architecture that focuses on low footprint and easy integration with other neural architectures. It has considerably fewer parameters than the models described in the previous section,

and it does not require a complex input in both training and execution (such as tables, preprocessed lists or programs traces). Additionally, unlike the previous models, the GRPU instructions can have any number of arguments, due to not requiring softmax selection, and of operations transforming those arguments in a single step.

3.1 The Architecture

Figure 1 exhibits the GRPU architecture, which is built upon the structure of a regular GRU. GRPU is not only easily exchangeable wherever a GRU can be used, enabling traditional algorithmic manipulation of it's inputs, but it can also be implemented with just a few lines of codes over the GRU. The fundamental difference between the two models is the way the new state is produced, but this small difference also affects how everything else is interpreted.

Thus, in GRPU, the affine transformation is replaced by an *Arithmetic and Logic Unit* (ALU), a module that executes one operation for each set of fields of the hidden state to produce the next state values. The *Virtual Machine* (VM) state, which replaces the hidden state in the GRU, is $h^{vm} \in \mathcal{R}^N$, where N is both the ALU's operation's outputs sizes summed and the argument's sizes summed. In other words, the VM state is both the arguments for the ALU, and the outputs of the ALU.

The ALU receives the previous VM state and returns a new candidate for the next state from the results of each operation. The reset gate, in this context, operates as the argument selector, responsible for determining which arguments will be fed to the ALU, turning the ones that should be ignored to zero. The update gate defines which operations have their results kept and which ones are ignored. In this last case, the previous values of the VM state are restored, and the operation is replaced by a NOP, *No Operation*. The algorithm is, therefore produced by producing the GRU gates $[u_t, r_t]$ based on the inputs, which is equivalent to producing the opcode $[operations, operands]_t$. Calculating every step gives the final algorithm, like the example displayed in Fig. 2.

Unlike with GRUs though, the hidden state, or the VM state, shouldn't be used in the creation of the gates output, and therefore in the creation of the instructions. This is done so the model can learn generic algorithms, that can automatically deal with data not seen in the training base. In the current

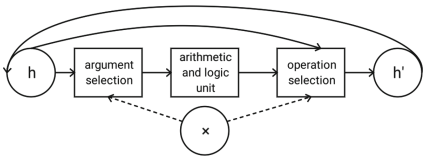

Fig. 1. The basic Gated Recurrent Programmer Unit. Dashed lines are the input of the gates, normal lines are the hidden (VM) state path.

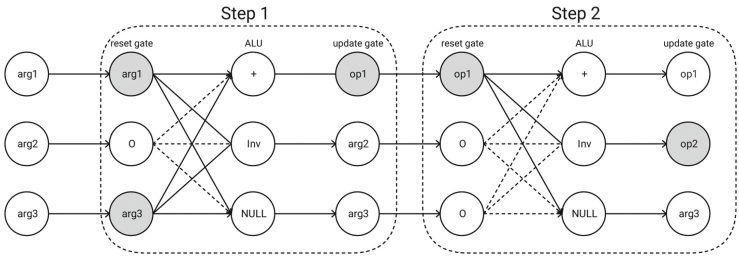

Fig. 2. Example of a two step algorithm: -(arg1+arg3). Each row has one argument and one operation throughout two recurrent steps. The reset gate selects the arguments for the ALU operations (grayed in the image with solid lines), while the update gate selects which operation results or arguments will be kept (grayed operation results).

architecture it means that there is a direct mapping between the current input and the respective instruction.

While this behavior is sometimes enough, we would like the model to use past information for creating the algorithm, and, for that reason, we include an additional *controller* unit, which acts in parallel to the programmer and has the same structure as the GRU. The complete model is depicted in Fig. 3, and represented by the following set of equations (from Eqs. 2 to 5):

$$r_t = \sigma(W_r.[h^c_{t-1}, x_t] + b_r) \tag{1}$$

$$u_t = \sigma(W_u.[h^c_{t-1}, x_t] + b_u) \tag{2}$$

$$\widetilde{h}^c_t = \tanh(W.[r^c_t * h^c_{t-1}, x_t] + b) \tag{3}$$

$$\widetilde{h}^{vm}_t[i] = ALU(r^{vm}_t, h^{vm}_{t-1}, external_t, operation[i]) \tag{4}$$

$$h_t = (1 - u_t) * h_{t-1} + u_t * \tilde{h}_t \tag{5}$$

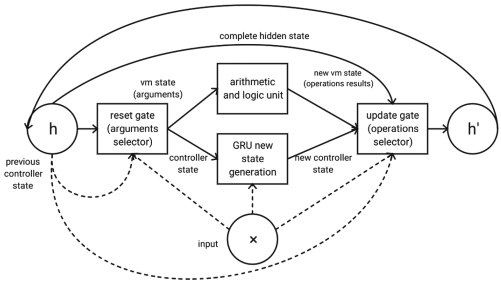

Fig. 3. The Gated Recurrent Programmer Unit. The upper part is the virtual machine, which executes the instruction according to the selections made by the gates. The lower part is the controller, which encodes a representation of all past inputs for the gates, producing instructions that aren't just a mapping of the current input.

Where vm defines the Virtual Machine (VM) section and c the controller section of the state and gate outputs, $h_t = [h_t^{vm}, h_t^c]$, $r_t = [r_t^{vm}, r_t^c]$ and $u_t = [h_t^{vm}, h_t^c]$ are the hidden state (formed by the concatenation of VM and controller states), reset gate (which assumes the task of argument selector for the VM state) and update gate (which assumes the task of the operation selector for the VM state), respectively. $\widetilde{h}_t = [\widetilde{h}_t^{vm}, \widetilde{h}_t^c]$ is the next state candidate. The ALU is a function that receives the VM state (arguments), the argument selection (reset gate output), any external data or differentiable memory that can be read/write through specific operations, and the list of operations to apply to the arguments.

3.2 The Arithmetic and Logic Unit (ALU)

The ALU natively supports n-ary operations, with the arguments selected directly with the argument selector. But one aspect that must be considered is what is the neutral element in the operation. The argument selector rejects arguments by multiplying them by zero. This behavior does not influence operations such as summation and the logical *or*. In other cases, though, such as the product or the logical *and*, a zero valued (rejected) argument would guarantee that the result is zero or False, respectively. To solve this issue, we introduce a transformation that makes rejected arguments (in which $r_t[i] = 0$) to have value one, instead of zero, and selected arguments to have the argument value itself, which may include zero. Table 1 shows the output we would like the both cases have.

Table 1. Target inputs for operations with neutral element 0 and 1.

Input (i)	Selector (r)	Neutral 0	Neutral 1	Input (i)	Selector (r)	Neutral 0	Neutral 1
0	0	0	1	x	0	0	1
0	1	0	0	x	1	x	x

An additional complication is that the argument selector gate is not restricted to binary outputs, but instead, covers the entire space between 0 and 1. To handle that we need to work on a superset of the Boolean algebra, like the Fuzzy Logic [6]. In particular, we choose the following generalized form for the basic logic operators, though other options are also possible: x AND $y = x * y$, x OR $y = 1 - (1 - x) * (1 - y) = x + y - x * y$ and NOT $x = 1 - x$.

Converting the neutral 1 column in terms of i and r in the truth Table 1 into a sum of products representation (where "." is the logical *and*, "+" is the logical *or*, and "$\overline{x}$" is the logical negation of x) we get $i.r + i.\overline{r} + \overline{i}.\overline{r}$. Next, by factoring $\overline{r}$ on the last two terms, we reach $\overline{r}.(i + \overline{i}) + i.r$, and by applying the identity $i + \overline{i} = 1$), we reach Eq. 6.

$$\overline{r} + i.r \tag{6}$$

Then, replacing the boolean operators for the fuzzy operators in the form of (NOT r) OR (i AND r), we get $(1-r)$ OR $(i*r) = (1-r)+(i*r)-(1-r)*(i*r) = 1 - r + i * r - i * r + i * r^2$, which brings us the Eq. 7.

$$1 - r + i * r^2 \tag{7}$$

Similarly, the sum of product form for the neutral 0 in Table 1 is simply $i\ AND\ r$, and, therefore in the generalized operators it is defined as $i * r$, which is already how the reset gate output is applied to the hidden state.

Thus, for any operation wherein the neutral element is zero we do $i * r$ and for any operation wherein the neutral element is one we apply Eq. 7 as its input.

For lesser arity operations, it's possible to simply eliminate some of the connections to the arguments (for example a toggle operation only needs a connection to it's previous result), and/or to use aggregate functions. By averaging the reset gate outputs before multiplying the VM state, it's also possible to have a soft selection equivalent to the softmax.

Besides the operations that map arguments to results, algorithms also require testing and flow control, and for that we first have to define *comparison operations*. Comparison operations typically have arity two (such as equal, not equal, less than, greater than), or one (equal to zero, not equal to zero, etc.) and return one if the condition is true, or zero otherwise. The way we implement the differentiable *not equal* (and the *equal*, by simply subtracting it from 1) is by having $|arg1 - arg2|/(|arg1 - arg2| + \epsilon)$ where ϵ is a constant to avoid division by zero. *Greater than* and $less - than$ can be implemented with a shifted sigmoid (logistic) function, approximating the Heaviside step function.

With the comparison operator, we can implement an element of control flow in the differentiable machine, the *conditional operation*. It makes the instruction to be executed only if the condition determined by a comparison operation, or a combination of them through logical operators, is met, and otherwise all the instruction is rejected. This is implemented by changing the operation selection mechanism according to Table 2, in which $\widetilde{u}_{cond}$ is the operation selector value (update gate value) for the conditional operation, $\widetilde{u}_{op}$ is the operation selector value for the target normal operation, h_{cond} is the result of the comparison used for the conditional, and u_{op} is the final operation selector values (the value of the operation or a NOP, or No Operation, equivalent to the update gate rejecting the operation). Simplifying the table like with the neutral element above:

$$u_{op} = \widetilde{u}_{op} \text{ AND } ((\text{ NOT } \widetilde{u}_{cond}) \text{ OR } \widetilde{h}_{cond}) \tag{8}$$

And using the same transformation inspired by Fuzzy Logic we discussed above, we arrive in the Eq. 9 below:

$$u_{op} = u_{op} * (1 + \widetilde{u}_{cond} * (\widetilde{h}_{cond} - 1)) \tag{9}$$

And for integrating it within the model equations, with u_t being the final output of the update gate for using in Eq. 5, $\widetilde{u}_t^c$ the controller section and $\widetilde{u}_t^{vm}$ the VM section of the update gate calculated in Eq. 2:

$$u_t = [\widetilde{u}_t^{vm}, \widetilde{u}_t^{vm} * (1 + u_{cond} * (\widetilde{h}_{cond} - 1))] \tag{10}$$

If the rejection condition happens, the whole programmer section of the update gate is multiplied by a scalar zero, and the new VM state becomes h_{t-1}^{vm}, and, therefore, the algorithm does not produce any effect in that step.

Table 2. Desired output when accepting or rejecting the input.

u_{cond}	$\widetilde{h}_{cond}$	$\widetilde{u}_{op}$	u_{op}	u_{cond}	$\widetilde{h}_{cond}$	$\widetilde{u}_{op}$	u_{op}
0 (-)	0 (-)	0 (-)	0 (nop)	1 (if)	0 (false)	0 (-)	0 (nop)
0 (-)	0 (-)	1 (do op)	1 (op)	1 (if)	0 (false)	1 (do op)	0 (nop)
0 (-)	1 (-)	0 (-)	0 (nop)	1 (if)	1 (true)	0 (-)	0 (nop)
0 (-)	1 (-)	1 (do op)	1 (op)	1 (if)	1 (true)	1 (do op)	1 (op)

3.3 Expanding the Model

Since the GRPU is similar in structure to a GRU, it can be extended in similar ways. For instance, by stacking a number of GRPUs it is possible to have different control flows, executing multiple operations per step, according to the number and order of transformations over the VM state. Another possibility is to use the encoder-decoder with soft attention [2] as inspiration, allowing the model to learn its own sequencing through the input, while also decoupling the input size from the program size.

4 Experimental Results

To produce the results presented here, we run all the tests with Tensorflow [1] on a single GPU, Adam optimization, learning rate 10^{-4}, and, otherwise, default parameters and no regularization. The controller hidden state has size 100.

4.1 The Adding Problem

To evaluate the potential to learn long algorithms, we use a variant of the RNN Adding Problem described in [13]. In each step the network is fed with a control value of either -1, 0 or 1 and an input value ranged $[0, 1]$. If the control is 1, which always happens in exactly two of the steps, then the corresponding input value should be one of the operands in the sum. There are between 50 and 55 steps. With a 10,000 samples training set and a 1,000 samples test set, batch size of 100, and using a bidirectional GRU with the outputs connected to a fully connected linear regression layer, the cited author achieves the mean squared error of 0.0041 on the test set.

Using the GRPU, we feed only the control vector to the controller unit to avoid dependence between the induction of the algorithm and the processed data. The ALU also contains 3 operations, a READ operator that returns the control vector, an ADD operator and a PRODUCT operator. This means that each step

has to choose to store the result of each of the 3 possible operations, or keep the previous argument, and to choose any combination of the 3 previous results as input for the operations, creating a very large search space with a program up to 55 instructions long. The output of the model is the result of the sum.

Table 3. Experiments. *Bidirectional GRU results from [13]

Configuration	1,000 epochs (training)	1,000 epochs (test)
Bidirectional GRU - batch 100 - 1,000 samples*	N/A	0.0041
GRPU - batch 100 - 10,000 samples	0.247	0.759
GRPU - batch 32 - 32,000 samples	0.0089	0.00699
GRPU - batch 10 - 10,000 samples	0.0000387	0.00709
GRPU - batch 10 - 10,000 samples - Varying number of steps	0.000426	0.000696
GRPU - batch 10 - 10,000 samples - (Multiplication Variant)	0.00616	0.0166
GRPU - batch 10 - 10,000 samples (Conditional Variant)	0.06	0.06

Table 3 shows that using the same batch size leads to very poor performance, indicating that the model is more prone to getting stuck in local minima. Either increasing the number of samples or reducing the batch size, which increases the stochastic effect, brings the results much closer to the more complex traditional model. Starting with just 10 steps and increasing the number up to the target throughout the epochs yields the best generalization.

4.2 Other Variations

Just changing the example above from addition to product, and changed the input range to $[0.5, 1.5]$, to prevent values frequently close to zero, allow us to evaluate the logic for the operations with neutral element one. The network behaved similarly, reducing the error to adequate levels after the 1,000 epochs, as seen on the Multiplication Variant on Table 3.

To test the conditional, we moved the control vector of the Adding Problem to the virtual machine, to be read on a second READ operator. It's also added a conditional operation that checks if it's input is 1, and if otherwise it forces a NOP in the step. This adds to 5 operations in the ALU, and the controller in this variation has no input besides it's state, and it's therefore incapable of choosing on it's own when to select the ADD operation and when to skip. This variation converges very fast, but gets easily stuck in a local minima worse than the original variant.

5 Conclusions and Future Work

Here, we presented a novel Neural Programming architecture that can help building a framework connecting neural networks and traditional programming. It has the potential of helping both models that write programs autonomously and users to integrate their logic within the neural network operation. The experiments have found some of the issues of previous neural programmer works: the convergence of such models is not trivial, possibly since the higher restriction on the search space may conduct to more local minima. More research in this area could provide better insights on the model behavior during training.

A number of further tests could be conducted in future works to better understand the potential of our model, such as tuning the hyper-parameters and ALU settings, adding regularization, experimenting with transfer learning and domain adaptation using the added transparency, evaluating deep GRPU models, and also techniques to extract efficient discrete algorithms.

References

1. Abadi, M., et al.: TensorFlow: large-scale machine learning on heterogeneous systems (2015). https://www.tensorflow.org/
2. Bahdanau, D., Cho, K., Bengio, Y.: Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473 (2014)
3. Bengio, Y., Simard, P.Y., Frasconi, P.: Learning long-term dependencies with gradient descent is difficult. IEEE Trans. Neural Netw. **5**(2), 157–166 (1994)
4. Cho, K., et al.: Learning phrase representations using RNN encoder-decoder for statistical machine translation. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, pp. 1724–1734. ACL (2014)
5. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)
6. Klir, G.J., Yuan, B.: Fuzzy Sets and Fuzzy Logic: Theory and Applications. Prentice Hall, Upper Saddle River (1995)
7. Kurach, K., Andrychowicz, M., Sutskever, I.: Neural random access machines. ERCIM News **2016**(107) (2016)
8. Maclaurin, D., Duvenaud, D., Adams, R.P.: Autograd: effortless gradients in numpy (2015)
9. Neelakantan, A., Le, Q.V., Sutskever, I.: Neural programmer: inducing latent programs with gradient descent. CoRR abs/1511.04834 (2015). http://arxiv.org/abs/1511.04834
10. Reed, S.E., de Freitas, N.: Neural programmer-interpreters. CoRR abs/1511.06279 (2015). http://arxiv.org/abs/1511.06279
11. Vinyals, O., Fortunato, M., Jaitly, N.: Pointer networks. In: Advances in Neural Information Processing Systems (NIPS 2015), vol. 28, pp. 2692–2700 (2015)
12. Xu, K., et al.: Show, attend and tell: neural image caption generation with visual attention. In: Proceedings of the 32nd International Conference on Machine Learning, pp. 2048–2057 (2015)
13. Zhou, G.B., Wu, J., Zhang, C.L., Zhou, Z.H.: Minimal gated unit for recurrent neural networks. Int. J. Autom. Comput. **13**(3), 226–234 (2016)