Searching for Activation Functions Using a Self-Adaptive Evolutionary Algorithm

Andrew Nader
Department of Computer Science and Mathematics,
Lebanese American University
Byblos, Lebanon
andrew.nader@lau.edu

Danielle Azar
Department of Computer Science and Mathematics,
Lebanese American University
Byblos, Lebanon
danielle.azar@lau.edu.lb

ABSTRACT

The introduction of the ReLU function in neural network architectures yielded substantial improvements over sigmoidal activation functions and allowed for the training of deep networks. Ever since, the search for new activation functions in neural networks has been an active research topic. However, to the best of our knowledge, the design of new activation functions has mostly been done by hand. In this work, we propose the use of a self-adaptive evolutionary algorithm that searches for new activation functions using a genetic programming approach, and we compare the performance of the obtained activation functions to ReLU. We also analyze the shape of the obtained activations to see if they have any common traits such as monotonicity or piece-wise linearity, and we study the effects of the self-adaptation to see which operators perform well in the context of a search for new activation functions. We perform a thorough experimental study on datasets of different sizes and types, using different types of neural network architectures. We report favorable results obtained from the mean and standard deviation of the performance metrics over multiple runs.

CCS CONCEPTS

• **Computing methodologies** → **Genetic programming**; **Genetic algorithms**; **Neural networks**;

KEYWORDS

Activation functions, Neural Architecture Search, Neural Networks, Deep Learning, Evolutionary Computation, Self-Adaptive Genetic Algorithms, Genetic Programming

ACM Reference Format:
Andrew Nader and Danielle Azar. 2020. Searching for Activation Functions Using a Self-Adaptive Evolutionary Algorithm. In *Genetic and Evolutionary Computation Conference Companion (GECCO '20 Companion), July 8–12, 2020, Cancún, Mexico.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3377929.3389942

1 INTRODUCTION

The ReLU activation function defined by $f(x) = max(0, x)$ is the most commonly used activation function nowadays, since it is fast to compute and allows for the training of deep networks. However, ReLU is not optimal and it does have some problems, such as the dying neurons problem [2]. Other modern activation functions include Leaky ReLU, ELU, PReLU, SELU, GELU, and Swish [3]. All of these activation functions, except for Swish, were designed by hand. Inspired by the recent success of Neural Architecture Search methods, we propose a self-adaptive evolutionary algorithm to automatically search for new activation functions that work well. We also look into the activation functions found to see if they have any common properties such as monotonicity.

2 ALGORITHM DESCRIPTION

We use a self-adaptive evolutionary algorithm which relies on genetic programming to search for new activation functions. In our approach, we encode our chromosome as a sequence of five genes of the form *<activation function, activation function crossover operator, activation function mutation operator, chromosome crossover operator, weight initialization scheme>*. The first gene consists of the activation function which is encoded as a standard genetic programming tree, with the leaf nodes being the input to the activation. The unary operations available are the predefined tensorflow functions *relu, elu, sigmoid, tanh, swish, sin, cosin, atan ,sinh, cosh, leaky relu, softplus, erf*, and *absolute value*. The binary operations we use are *add, subtract, multiply, maximum*, and *minimum*. Instead of fixing the choice of mutation and crossover operator for the activation function trees, we encode them in a self-adaptive way as part of our chromosome. The way two chromosomes can be crossed over is also subject to self-adaptation. Given two parent chromosomes, we first randomly choose a chromosome crossover operator from one of the parents, and we apply it to produce two children chromosomes. We then randomly choose one of the activation function crossover operators present in the two parent chromosomes, and we use it to crossover the activation functions present in the children. Finally, the mutation operator of each child chromosome is used to mutate the activation function with a certain probability. We chose a self-adaptive scheme since we have no prior knowledge as to which variation operators are best when dealing with activation functions, and because there is evidence that the best choice of operator is not fixed, but that it varies depending on which generation the algorithm is on. We chose to use the DEAP python library for the implementation of the variation operators [1]. The tree crossover operators available to our algorithm are the built in DEAP *cxOnePoint* and *cxOnePointLeafBiased* (with termpb set to 0.9) The tree mutation operators available are also available in DEAP, and they are the *mutShrink*, *mutInsert*, and *mutNodeReplacement* operators. We constrain the tree size to be between 4 and 10 using a static

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
GECCO '20 Companion, July 8–12, 2020, Cancún, Mexico
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7127-8/20/07…$15.00
https://doi.org/10.1145/3377929.3389942

bloat limit. The chromosome crossover methods are the standard one-point-crossover with random index, two-point-crossover with random index, and uniform crossover. Instead of also self-adapting the mutation and crossover rates, we chose to keep them constant at 5% and 80% respectively, since we hypothesize that the choice of operators will play a more important role here. We also add a weight initialization gene: the possible weight initialization methods for our algorithm are *random normal, random uniform, truncated normal, variance scaling, orthogonal lecun uniform, lecun normal, glorot uniform, glorot normal, he normal*, and *he uniform*. We use the ADAM optimizer with a batch size of 32 for all of our experiments.

We test our algorithm on three datasets: the German Credit dataset, the NSL-KDD dataset, and the CIFAR-10 dataset. For the German Credit dataset, we keep a constant population size of 500 for 200 generations, with each individual trained for 50 epochs. The NSL-KDD and CIFAR-10 datasets are very large, and due to a lack of access to computational power, we choose to take advantage of our observations that the relative ranking of individuals stays the same, and we gradually decrease the size of the population while increasing the number of epochs. We start with 500 individuals trained for 10 epochs, decrease to 250 individuals trained for 20 epochs at the second iteration, 125 individuals trained for 30 epochs at the third iteration, and then 100 individuals trained for 50 epochs at the fourth iteration. This is kept up to the 20th iteration, where we start to decrease the population size by 10 at each iteration while keeping the number of epochs constant, until we reach a population size of 30 at iteration 29. We then run one more iteration with a population size of 30 and an epoch number of 50, and we stop the evolutionary algorithm. Finally, we get the mean and standard deviation of the performance metrics of the final individuals over 50 runs. The fitness of an individual in the genetic algorithm is evaluated on a randomly generated development set (10% for the NSL-KDD and CIFAR-10, 30% for the German Credit) to prevent finding activations that over-fit easily. We use the f1-measure for the German Credit dataset, accuracy+f1 measure for the NSL-KDD dataset, and accuracy for the CIFAR-10 dataset. When evaluating on the test set, we use model-checkpointing for 100 epochs on a development set for the NSL-KDD and CIFAR-10 datasets: we do not do this for the German Credit dataset since it is small and thus the development set results in a high variance metric. The test sets for the NSL-KDD and CIFAR-10 are predefined, and we chose to use a 2/3-1/3 split for the German Credit dataset. We use an architecture of 2 hidden layers and 30 neurons each for the German Credit, and 4 hidden layers of 100 neurons for the NSL-KDD. Both architectures use L_2 regularization and a dropout rate of 50%, and we use MinMax scaling for both. We use the LeNet-5 architecture for the CIFAR-10, and we scale by dividing the pixels by 255.

3 EXPERIMENT RESULTS

All three functions in Table 1 were associated with a Random Uniform weight initialization scheme. The activation functions do not appear to share similar traits other than the fact that they are monotonic.

The evolutionary algorithm associated atan(x+|x|+||x|+x|) in Table 2 with a truncated normal weight initialization scheme, and the other 2 individuals with a glorot normal weight initialization scheme. One of the activation functions is not monotonic, which suggests that this is not a necessary property, and two functions have hard zeroes like ReLU.

Table 1: Results on German Credit Dataset

	Accuracy	Recall	Precision	F1 Measure
2x+cos(x)	0.72982(0.01035)	0.7994(0.02984)	0.81762(0.00956)	0.80797(0.01098)
6x+2cos(x)+max(2x,x)+max(x,sin(cos(x))	0.73976(0.01413)	0.82145(0.03619)	0.81515(0.01074)	0.81768(0.01408)
6x+sinh(2x)+sinh(x)	0.73679(0.00927)	0.83123(0.01587)	0.80559(0.00978)	0.81807(0.00712)
ReLU	0.722(0.00863)	0.76826(0.01422)	0.82896(0.00757)	0.79735(0.00754)

Table 2: Results on NSL-KDD Dataset

	Accuracy	Recall	Precision	F1 Measure
atan(x+|x|+||x|+x|)	0.77854(0.01847)	0.9241(0.00283)	0.67906(0.02144)	0.78263(0.01375)
swish(max(sin(erf(x)),x)+|elu(x)|)+|x+min(x,erf(x))|	0.77287(0.01122)	0.92428(0.00203)	0.67206(0.01293)	0.77816(0.00832)
erf(relu(x))+||x|+min(x,erf(min(2|x|,|x|+swish(x))))|	0.77974(0.01836)	0.9242(0.00255)	0.68041(0.02124)	0.78357(0.01369)
ReLU	0.74662(0.00616)	0.92786(0.00173)	0.64267(0.00657)	0.75935(0.00414)

Table 3: Results on CIFAR-10 Dataset

	Accuracy
max(|x|,x^2)	0.65736(0.0066)
max(|x|,x^3)	0.65009(0.00567)
|x|	0.64233(0.0055)
ReLU	0.63184(0.01136)

The algorithm chose the Orthogonal weight initialization method for all three activations in Table 3. The activations are non monotonic, they are strictly positive, and they do not have a hard zero. We have succeeded in finding activations that outperform ReLU by a meaningful margin in all of our experiments: this suggests that it is worth evolving new activation functions when performing Neural Architecture Search. However, the variation operators that were most prominent in the individuals were different for each dataset: for example, cxOnePointLeafBiased completely disappeared from the population in the experiment on the NSL-KDD run, while it remained in a good percentage of the individuals on the German Credit dataset (around 40%). There are three possible explanations for this phenomenon: First, the variation operators do not play as big a role as assumed. Second, it could be that the self-adaptiveness did not have the time to take hold on two of the experiments as they were run for 30 epochs only. Third, the best variation operators are problem dependent. For future work, we will try out our algorithm on bigger architectures and study the effect of variation operators on activation functions in more depth.

REFERENCES

[1] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: Evolutionary Algorithms Made Easy. *Journal of Machine Learning Research* 13 (jul 2012), 2171–2175.
[2] Lu Lu, Yeonjong Shin, Yanhui Su, and George Em Karniadakis. 2019. Dying ReLU and Initialization: Theory and Numerical Examples. *arXiv preprint arXiv:1903.06733* (2019).
[3] Prajit Ramachandran, Barret Zoph, and Quoc V Le. 2017. Searching for activation functions. *arXiv preprint arXiv:1710.05941* (2017).