STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXI**, Number 2, 2016

A GENETIC ALGORITHM APPROACH FOR EVOLVING NEURAL NETWORKS

ALEXANDRU-ION MARINESCU

Abstract. We present an alternative approach for training feed-forward neural networks (abbrev. NN) by means of a genetic algorithm (abbrev. GA) that alters the network's hidden weights and biases. We, by no means, out-rule the back-propagation training algorithm, but instead use it to train the evolved NNs for a much smaller number of generations and focus more on the mutation and crossover operators and how they can be applied.

The basic principle involved is that each and every NN can be treated as a chromosome for a GA and, as a consequence, is subject to the mutation and crossover operators. A notable advantage of our approach is that we not only avoid over-fitting the NN, but are also able to alter the number of hidden neurons that make up the hidden layer of the NN, effectively removing the need for the user to specify them explicitly. We manage to outperform plain-vanilla NNs by a factor of 1 to 10 percent on well known data sets. At first this may not seem significant, but it becomes crucial when dealing with applications where accuracy is critical and training time is not an issue (such as disease diagnosis).

1. Introduction

The following paper assumes that the reader has a fair understanding of how feed-forward NNs and evolutionary algorithms work. The rationale behind our decision to evolve a NN by means of a GA stems from the fact that a lot of trial and error is involved in fine tuning the so called "hyper-parameters" (i.e. number of hidden neurons and NN learn rate) such as to maximize accuracy on training data and minimize error on test data. Consequently, by applying the standard GA operators (mutation and crossover) on our NNs we are able to achieve the same accuracy of a plain-vanilla NN with ten times less

Received by the editors: October 15, 2016.

2010 *Mathematics Subject Classification.* 92B20.

1998 *CR Categories and Descriptors.* A.1 General Literature [**INTRODUCTORY AND SURVEY**]; I.2.6 Computing Methodologies [**ARTIFICIAL INTELLIGENCE**]: Learning – *Connectionism and neural nets*.

Key words and phrases. biology, AI, neural network, genetic algorithm, evolution, mutation, crossover, optimization.

number of training epochs, with the added benefit of completely delegating the training process to a separate task. In turn, this parallel training model enables us to have a generous chromosome pool of NNs while lowering the performance footprint.

The main phenomenon that acts in the detriment of NNs, which all implementations wish to avoid is known in literature as "over-fitting" and it roughly means that the NN has become attuned to the training data so well that it is incapable of correctly classifying the test data, or equivalently in mathematical terms, it is unable to escape a local minimum/maximum. To overcome this issue, we have altered the classical GA implementation and propose a novel means for refreshing the chromosome pool with new candidates for genetic material. The main trade-off that we wish to point out is that we significantly improve accuracy at the expense of execution time.

2. Conceptual presentation

Our GA [3, 7, 4] implementation is a modified version of its classical counterpart and has an additional mechanism for dealing with over-fitting. It monitors the fitness trend of the chromosomes from the internal chromosome pool and if it detects no significant variation triggers a "cataclysm" during which a percentage of the fittest chromosomes is destroyed and replaced with a fresh new batch of individuals. At first, this may seem counter-intuitive, but we still keep a copy of the best chromosomes in what we call the "garden of Eden". Finally, we sort the garden of Eden with the fittest chromosome being the returned solution.

The next step we took was to treat a three layer (i.e. input, hidden and output) feed-forward NN [9] as a chromosome within a GA, consequently subjecting it to the mutation and crossover operators [8, 6]. This is achieved easily in a modern OOP language, such as C# by having an interface IChromosome with the following methods: "Mutate" which takes as input an IChromosome and returns an IChromosome, and "Crossover" which receives two IChromosome parameters and outputs the corresponding pair of IChromosome offspring.

Afterwards all that remains to be done is inherit the NN from the IChromosome interface and implement the corresponding methods. The reader may ask whether the evolved NN is used as-is, directly from the GA, for which the answer is no. After successful mutation/crossover, the NN is still trained, albeit for a much smaller number of epochs (the training algorithm we applied is the standard back-propagation training). The workload of training the chromosomes is split between the available processor cores using TPL (Task Parallel Library) in order to improve overall performance.

The concept of training a NN using a GA stemmed from the following research involving Continuous-Time Recurrent Neural Networks [2, 1], which are notoriously difficult to train using the classical back-propagation algorithm and thus the authors suggest an alternate training scheme, which involves employing a GA [8]. The original contribution of this paper to the field of AI is the parallel execution of the training method, together with a specialized version of the GA and possibility to offload the most intensive tasks to the GPU, which will be detailed in future research.

The next section focuses mainly on aspects regarding our implementation of a GA, detailing how each parameter affects the overall behavior of the GA, along with the implications of inheriting the NN from the IChromosome interface in order for it to function within a GA. Afterwards we discuss the benchmarks we used for testing the accuracy of our combined GA-NN approach for solving concrete classification tasks. We conclude the paper by stating the future directions of our research in the field of AI.

3. NNs in the context of GAs

In our particular case, the parameters used for evolving a NN are as follows: LogEnabled = false (disable logging of events for faster execution), PoolSize = 50 (the number of chromosomes in the pool is capped at 50), PoolSortMode = PoolSortMode.Descending (sort the pool in descending order with respect to fitness), ProbabilityOfMutation = 0.1 (mutation is performed in 1 of 10 chromosomes), ProbabilityOfCrossover = 0.1 (crossover is performed in 1 of 10 chromosomes, with a random, more fit partner), ThresholdForCataclysm = 0.001 (cataclysm is performed if the current pool weight minus the previous pool weight in absolute value is smaller than this value), PercentageOfCataclysm = 0.25 (25% of the fittest chromosomes are replaced with new genetic material in the event of a cataclysm).

The pool weight which we have mentioned above is computed in this fashion (index is zero-based):

```
double WeightChromosome(double min,double max,int index)
{
  if(Abs(max-min)<Epsilon)
    return 1/(index+1);

  return (pool[index].Fitness()-min)/(max-min)/(index+1);
}

double WeightPool()
{
  double min=-Infinity, max=+Infinity;

  if(PoolSortMode==Descending)
  {
    max=pool[0].Fitness();
    min=pool[PoolSize-1].Fitness();
  }

  if(PoolSortMode==Ascending)
  {
    min=pool[0].Fitness();
    max=pool[PoolSize-1].Fitness();
  }

  double weight=0;
  for(int i=0;i<PoolSize;i++)
    weight+=WeightChromosome(min,max,i);
  return weight;
}
```

As the reader may have noticed, the weight of a chromosome is directly proportional to the minimum and maximum fitness of the pool and its index order in the sorted pool. The total weight of the pool is the sum of the weights of all its chromosomes.

The core of the GA is the evolutionary iteration step, which is run for a given number of generations (100 in our particular case):

```
void EvolveOne()
{
  for(int i=0;i<PoolSize;i++)
  {
    if(R.NextDouble()>ProbabilityOfMutation)
      continue;
    pool.Add(pool[i].Mutate());
  }

  for(int i=1;i<PoolSize;i++)
  {
    if(R.NextDouble()>ProbabilityOfCrossover)
      continue;
    pool.Add(pool[i].Crossover(pool[R.Next(i)]));
  }

  Task.WaitAll(tasks);
  tasks.Clear();

  pool.Sort();
  if(PoolSortMode==Descending)
    pool.Reverse();
  pool.Trim(PoolSize);
  gardenOfEden.Add(pool[0]);

  poolWeightPrev=poolWeightCrt;
  poolWeightCrt=WeightPool();
  if(Abs(poolWeightCrt-poolWeightPrev)<ThresholdForCataclysm)
    Initialize(PercentageOfCataclysm);
}
```

Notice the "Task.WaitAll" construct. This notifies the GA that the resulting offspring NNs are still training and will be ready for the current iteration after this instruction yields (it is part of the TPL library which we have mentioned earlier). We do not leave the resulting offspring untrained, but instead train them on a much smaller number of epochs (10 vs. 100).

The fitness of a NN is computed as the difference between its accuracy on training data and error on test data (we apply an 80% training and 20% testing scheme):

- The accuracy is computed on the training data, equal to $\frac{correct}{correct+wrong}$;
- The error is computed on the test data, equal to the sum of the squared errors for each sample input;

Internally, the NN stores several arrays of weights and biases which are updated via back-propagation during each epoch. Of particular interest to us are the "Input-Hidden Weights" and the "Hidden-Output Weights" matrices and the "Hidden Biases" array respectively.

We have settled upon a mutation operator which is either deleterious or additive. The decision regarding which type will be applied is made randomly, whilst also taking into consideration some lower and upper bounds for the hidden neuron count (1 and 50 respectively in our case). The deleterious mutation chooses a random hidden neuron index and deletes the corresponding neuron, effectively removing its associated weights and biases from the Input-Hidden Weights, Hidden Biases and Hidden-Output Weights arrays. We have left room for improvement here, suggesting that the choice of which neuron is deleted can be made not randomly, but based on how well it behaves during the algorithm (such as successful activation count). Similarly, the additive type of mutation inserts a new neuron in the NN and updates the three structures we mentioned earlier. The new neuron is initialized randomly with some small positive value in the range $[10^{-4}, 10^{-3}]$. Lastly, the crossover operator takes as parameters two chromosomes (i.e. NNs), chooses two hidden neuron indexes that define a sub-sequence and are valid for both NNs (take into account the fact that crossover could be applied to NNs with a different number of hidden neurons) and exchanges this sub-sequence, together with the corresponding weights and biases between the two neural nets. One final aspect to keep in mind which is essential to the correct functioning of a NN is data normalization within the working domain of the activation function (in this particular case we used the hyperbolic tangent $f(x) = tanh(x) = \frac{2}{1+e^{-2x}} - 1$ which handles data in the range [-1,1].

4. Numerical results

We have compared our results against a plain 3-layer feed-forward NN consisting of fully-connected input, hidden and output layers, with fixed hidden neuron count and trained via back-propagation. Our candidate for testing is the NN we described above, which was evolved using a GA.

Data set description:

- Abalone [10] — Predicting the age of an abalone (a type of marine snail) from physical measurements. The age of abalone is determined by cutting the shell through the cone, staining it, and counting the number of rings through a microscope. Number of instances – 4177, number of properties – 9 (Sex, Length, Diameter, Height, WholeWeight, ShuckedWeight, VisceraWeight, ShellWeight, Rings);
- Balance [11] — This data set was generated to model psychological experimental results. Each example is classified as having the balance scale tip to the right, tip to the left, or be balanced. Number of instances – 625, number of properties – 5 (ClassName, LeftWeight, LeftDistance, RightWeight, RightDistance);
- Banknote [12] — Data were extracted from images that were taken from genuine and forged banknote-like specimens. For digitization, an industrial camera usually used for print inspection was used. The final images have 400 by 400 pixels. Due to the object lens and distance to the investigated object gray-scale pictures with a resolution of about 660 dpi were gained. Wavelet transform tools were used to extract features from images. Number of instances – 1372, number of properties – 5 (Variance, Skewness, Curtosis, Entropy, Class);
- Car [13] — Car evaluation database was derived from a simple hierarchical decision model originally developed for the demonstration of DEX. Number of instances – 1728, number of properties – 7 (BuyPrice, MaintPrice, DoorCount, PersonCount, LuggageBoot, Safety, Evaluation);
- Haberman [14] — The dataset contains cases from a study that was conducted between 1958 and 1970 at the University of Chicago's Hospital on the survival of patients who had undergone surgery for breast cancer. Number of instances – 306, number of properties – 4 (PatientAgeAtOp, PatientYearOfOp, PositiveAxillaryNodes, SurvivalStatus);
- Iris [15] — This is perhaps the best known database to be found in the pattern recognition literature. The data set contains 3 classes of 50 instances each, where each class refers to a type of iris plant. One class is linearly separable from the other 2; the latter are not linearly separable from each other. Number of instances – 150, number of properties – 5 (SepalLength, SepalWidth, PetalLength, PetalWidth, Class);

- Letter [16] — The objective is to identify each of a large number of black-and-white rectangular pixel displays as one of the 26 capital letters in the English alphabet. The character images were based on 20 different fonts and each letter within these 20 fonts was randomly distorted to produce a file of 20,000 unique stimuli. Each stimulus was converted into 16 primitive numerical attributes (statistical moments and edge counts) which were then scaled to fit into a range of integer values from 0 through 15. Number of instances – 20000, number of properties – 17 (Letter, BoxPosHorizontal, BoxPosVertical, BoxWidth, BoxHeight, PixelCount, PixelsBoxX, PixelsBoxY, VarianceX, VarianceY, Correlation, MeanXXY, MeanXYY, EdgeCountLR, EdgeCorrelationXY, EdgeCountBT, EdgeCorrelationYX);
- Nursery [17] — Nursery Database was derived from a hierarchical decision model originally developed to rank applications for nursery schools. It was used during several years in the 1980's when there was excessive enrollment to these schools in Ljubljana, Slovenia, and the rejected applications frequently needed an objective explanation. The final decision depended on three sub-problems: occupation of parents and child's nursery, family structure and financial standing, and social and health picture of the family. The model was developed within expert system shell for decision making DEX. Number of instances – 12960, number of properties – 9 (ParentsOccupation, ChildNursery, FamilyForm, Children, Housing, Finance, SocialConditions, HealthConditions, Decision);

For all considered data sets the type of problem is classification for multi-class with balanced training data, transforming the output data into a discrete domain. The numerical results illustrate both the accuracy on training data and error on test data.

Plain NN						
Tester	Input	Hidden	Output	Fitness	Accuracy	Error
Abalone	10	7	1	0.52	1.00	0.48
Balance	4	5	3	0.75	0.87	0.12
Banknote	4	5	2	0.95	0.97	0.02
Car	21	13	4	0.97	0.98	0.01
Haberman	3	5	2	0.33	0.72	0.38
Iris	4	5	3	0.84	0.90	0.06
Letter	16	33	26	0.51	0.77	0.26
Nursery	27	31	5	0.92	0.97	0.05

TABLE 1. Benchmark results for a plain-vanilla NN.

GA NN						
Tester	Input	Hidden	Output	Fitness	Accuracy	Error
Abalone	10	7	1	0.52	1.00	0.48
Balance	4	15	3	0.98	1.00	0.02
Banknote	4	17	2	0.99	1.00	0.01
Car	21	24	4	0.99	1.00	0.01
Haberman	3	10	2	0.41	0.75	0.33
Iris	4	8	3	0.97	1.00	0.03
Letter	16	49	26	0.71	0.86	0.15
Nursery	27	44	5	0.99	0.99	0.00

TABLE 2. Benchmark results for a GA-trained NN.

After running the benchmarks through a total of 10 independent tests and selecting the best candidates in terms of fitness, our approach yields an increase in overall fitness by a factor of 1 to 10 percent (Tables 1 and 2). The abalone data set, as stated in literature, proves to be particularly challenging as a classification task [5]. Please remark that $Fitness = Accuracy - Error$, where Accuracy is the accuracy on training data and Error is the error on test data. It is very important to keep in mind that, although the NNs from the chromosome pool are trained for a much smaller number of generations than the standalone neural net and we utilize the CPU cores to their maximum potential, it is still orders of magnitude slower due to the amount of computations involved. Nevertheless, the results obtained point out that training NNs via a GA proves useful when high accuracy and low error are required, regardless of the time needed to train the NN, such as applications where the two factors are critical.

5. Conclusions

In the closing section of this article, we hope that we have provided valuable insight into how one can apply the benefits of a GA to classical feed-forward NNs in order to achieve more accurate results. The original contributions of this paper consist of a hybrid NN GA approach with a specialized GA that tries to avoid over-fitting, with the ability to auto-tune NN hyper-parameters. There is still a lot of room left for improvement, for example analyzing more advanced mutation/crossover schemes, but this is left to the reader's choice. Henceforth, we will focus on deep-learning algorithms, namely recurrent NNs and analyze how a GA can be applied to strengthen their overall performance. Additionally, we will explore the benefits of the recent advances in general-purpose GPU computing (GPGPU) and try to apply them to our framework.

References

[1] G. Bailador, D. Roggen, G. Tröster, G. Triviño, Real time gesture recognition using Continuous Time Recurrent Neural Networks, Proceedings of the ICST 2nd International Conference on Body Area Networks, BodyNets '07, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007.

[2] R. D. Beer, On the Dynamics of Small Continuous-Time Recurrent Neural Networks, Adaptive Behavior 3(4), 1995, pp. 469-509.

[3] L. Davis, Handbook of Genetic Algorithms, Van Nonstrand Reinhold, New York, 1991.

[4] D E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley Publishing Company, Inc., 1989.

[5] M. Keijzer, Genetic Programming: 7th European Conference, EuroGP 2004, Coimbra, Portugal, April 5-7, Proceedings, Volume 7, Springer Science & Business Media, 2004.

[6] P. Koehn, Combining Genetic Algorithms and Neural Networks: The Encoding Problem, Dissertation Thesis, University of Tennessee, December 1994.

[7] M. Mitchell, An Introduction to Genetic Algorithms, MIT Press, 1998.

[8] D. J. Montana, L. Davis, Training Feedforward Neural Networks Using Genetic Algorithms, Proceedings of the 11th International Joint Conference on Artificial Intelligence, Volume 1, IJCAI'89, 1989, pp. 762-767.

[9] R. Tadeusiewicz, R. Chaki, N. Chaki, Exploring Neural Networks with C#, CRC Press, Taylor & Francis Group, 2014.

[10] UCI Machine Learning Repository, Abalone Data Set, `http://archive.ics.uci.edu/ml/datasets/Abalone`.

[11] UCI Machine Learning Repository, Balance Scale Data Set, `http://archive.ics.uci.edu/ml/datasets/Balance+Scale`.

[12] UCI Machine Learning Repository, Banknote Authentication Data Set, `http://archive.ics.uci.edu/ml/datasets/banknote+authentication`.

[13] UCI Machine Learning Repository, Car Evaluation Data Set, `http://archive.ics.uci.edu/ml/datasets/Car+Evaluation`.

[14] UCI Machine Learning Repository, Haberman's Survival Data Set, `http://archive.ics.uci.edu/ml/datasets/Haberman%27s+Survival`.

[15] UCI Machine Learning Repository, Iris Data Set, `http://archive.ics.uci.edu/ml/datasets/Iris`.

[16] UCI Machine Learning Repository, Letter Recognition Data Set, `http://archive.ics.uci.edu/ml/datasets/Letter+Recognition`.

[17] UCI Machine Learning Repository, Nursery Data Set, `http://archive.ics.uci.edu/ml/datasets/Nursery`.

Department of Computer Science, Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Kogălniceanu 1, 400084 Cluj-Napoca, Romania

E-mail address: `amarinescu@cs.ubbcluj.ro`