Universitat Politècnica de Catalunya

Computational Intelligence

Neural Networks with Evolutionary Algorithms

Authors:

Benjamí Parellada
Clara Rivadulla

Fall Term 2022/2023

Contents

1 Introduction

Evolutionary neural networks are an adaptive approach that combines the adaptive mechanism of Evolutionary Algorithms (EAs) with the learning mechanism of Artificial Neural Network (ANNs). The use of EA approaches brings benefits to the usual way of derivative based methods of training. Specifically, the error function does not need to be continuous nor differentiable, it can be noisy, and it can incorporate discrete information. In the literature, there are many ways to combine the evolution with ANN, in this project we will focus on the following kinds of evolution:

- Evolution of the weights of the neural network. Where we will use EAs to find the actual parameters of the neural network instead of the usual backpropagation algorithm.
- Evolution of the architecture. Where we will use EAs to find the best actual architecture parameters of the neural network. This is usually a contentious topic in the creation of ANNs, since the architecture will have an impact on the neural net's performance. Specifically, we will fix the neural network to only have one layer and modify the amount of nodes it has. Moreover, we will also try and see if the neural network works better with linear or non-linear output function.
- Evolution of hyperparameters. In this case, we will only study the effect of the weight decay of the neurons, where we will see if EAs bring any benefit.

In order to achieve the project's deadline, some considerations have been taken, for example the use of synthetic data and the use of only one validation set.

2 Data Generation

Using synthetic data allows us to determine the sample size used for training, validation, and testing. Moreover, we can define the true generalization error of a model, the amount of noise, and the problem hardness.

We have decided to create a data set related to the performance of students in the *Bachelor Degree in Informatics Engineering* of the UPC. Specifically, we want to predict the grade that they will obtain in the subject of *Programming Projects* according to their previous results in two related subjects: *Programming 1* and *Programming 2*.

First, we list down all variables we want to include in the data set, and describe the requirements of each one. The following table presents the description, and some of the liberties we have taken in order to simplify the generation to make it concordant.

Variable	Variable Name	Description
Student ID	id	Unique identifier of a student.
Programming 1 Grade*	prog1_grade	The grade that the student got in the Programming 1 subject. The average grade for all students is 6.4 out of 10.
Programming 2 Grade*	prog2_grade	The grade that the student got in the Programming 2 subject. The average grade for all students is 6.7 out of 10.
Class Absences	absences	The number of absences of a student (number of days they've not attended to class) during their first year. Students tend to have an average of 5 absences.
Programming Projects Grade*	prog_proj_grade	The grade that a student obtained in the subject. The average mean grade is 7.3 out of 10.

* Variables that are correlated with one another

Table 1: Description of the variables included in our dataset.

We generate a dataset of $N = 3000$ data instances. The grades are clamped between 0 and 10.

In the generating process, we take into account that:

- The number of 'absences' is distributed according to a Poisson distribution $abscenses \sim Pois(\lambda = 5)$.
- For each grade, we will assume that it comes from a Normal Distribution. However, we want them to be correlated between themselves, hence, we shall use a multivariate normal distribution to generate the values of the grades. Thus, we need a vector of means and a variance-covariance matrix to generate our multivariate normal distribution. This is not a trivial matter, and it takes carefulness to create a covariance matrix randomly. To do so, we create a random matrix X and get $\Sigma = X^\top X$ as the covariance matrix. The variance-covariance matrix is generated randomly, however, we added two parameters: hardness and noisiness.
  - *Hardness*: is a parameter that controls the difficulty of the problem and its value should be in $[0, 1]$. The meaning of the parameter and its range are inverted, thus, a hardness of 0 actually makes the problem harder. This difficulty is achieved by multiplying the correlation part of the covariance matrix by the hardness. The reasoning behind this is that we expect the problem to be easier when the features are correlated with the target, and harder when there is no correlation.
  - *Noisiness* is used to scale the variance of the covariance matrix in order to be more accentuated or less. Be careful as these two parameters can actually make the previous matrix not positive semidefinite and make the generator fail.
- Moreover, to add a bit more complexity to the grade of *Programming Projects*, we multiply the generated value of the target feature a random value obtained from a normal distribution centered on the number of absences the student had multiplied by *(1 - hardness)*, and standard deviation *noisiness*. This is done only if hardness is not 1, and the value is also scaled accordingly to not create exaggerated values. This is done to create a nonlinear relationship between absences and the target feature on harder problems.
- To add even more complexity, we add a random value from a normal distribution with mean zero and standard deviation equal to the noisiness parameter.

Thus, there are many sources of randomness that will define the generalization error. Given p_1, p_2, p_p, and a, where each represents a variable, p_1 programming 1, p_p programming projects, a absences from sampled from the corresponding distributions. Then, if the hardness is not 1, the target feature is calculated as:

$$y = p_p \cdot a \frac{1 - hardness}{5} + \epsilon$$

where ϵ is random noise added, and 5 is a scaling factor found by trial and error. However, if hardness is equal to 1, we calculate it as:

$$y = p_p + \epsilon$$

This is how the data is generated, however, we will try to predict it with the following features, following R's model notation:

$$\hat{y} \sim p_1 + p_2 + a$$

Originally, we clamped the values to be a correct grade between 0 and 10, however, after trying to add complexity to the model, we decided to remove the clamp. This was done to have the frameworks result in larger differences in their score, otherwise, since the values were clamped into a small range, the RMSE was not very different independent of the hardness of the problem. Thus, due to the addition of the hardness, the resulting dataset cannot really be said to represent the grades, but it is still valid to analyze and infer how the different systems work. Instead of thinking about it as grades, we can think of it as life expectancy subtracted by being at the UPC.

2.1 Data Split

We generate $N = 3000$ instances of data which is then split into train, validation, and test sets according to the usual 80/10/10% split. The training will only be used to train the neural networks, either through backpropagation or evolutionary algorithms. The validation set will be used to select which architecture returns the best performance in form of the RMSE. Finally, the test set will be used to evaluate the performance on a never seen dataset which will return an unbiased metric of the error.

3 Methodology

The following sections present the methodology done to train the neural networks using Evolutionary Algorithms, both in training the weights and finding the best architecture parameters. The work done can be split into three groups depending on how the weights of the neural network were learned: derivative based, genetic algorithm, or CMA-ES. On top of this, there is a layer that will try and find the best architecture, either using Genetic Algorithms or CMA-ES.

Since the target feature is a continuous real value, we will be using regression to predict it. To evaluate the fit of the neural networks on the validation or test sets, we will predict the values $\hat{y}$ and compare these predictions to the actual value y, using the Root Mean Square Error (RMSE). This value by itself does not inform us much, however, we can

compare it with other models and the lower it is, the better. We can calculate it as:

$$RMSE = \sqrt{\sum_{i=1}^{n} \frac{(\hat{y}_i - y_i)^2}{n}}$$

3.1 Training Networks

1. **Derivative based**: the method consists of training a neural network using backpropagation. Each time a new architecture is presented, a new neural network with randomized weights is created. This network is trained for 100 epochs to fit the training data. Once the data is fit, we compute the RMSE from the predictions on the validation set to give an estimate on the error. The commented procedure was done for each architecture presented. To find what is the best architecture, we use Genetic Algorithms as explained in 3.2. The proposed methodology was inspired from [1], in which each architecture was tuned for only 5 epochs and the top 20 candidates were trained for more epochs to get a better fit. We did not need to use only 5 epochs, since our architecture is much simpler than the CNN they proposed in the paper. However, after selecting the best architecture, we will train it for more epochs.

2. **Genetic based**: the method consists of creating a vector of real valued weights, and instead of using backpropagation, use Genetic Algorithms to train the neural network. Each position of this vector represents the weight of the input layer to the hidden layer plus the bias, as represented in Figure 2, and these weights will be mutated and crossed at each evolution. This vector is initiated with random weights, which do not need to conform to the usual uniform distribution of values between $[-1, 1]$. Given the architecture – which will be found by Genetic Algorithms as explained in 3.2 – a neural network is constructed, and these weights are used to predict the validation set in order to get the RMSE and direct the fitness function, which we want to minimize. The proposed methodology was inspired from [2]

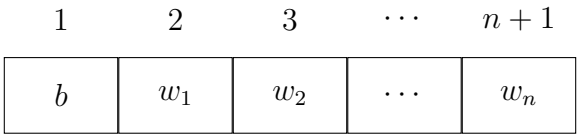

Figure 1: Representation of the real-valued vector of weights for a hidden layer of size $|\mathbf{w}| = n$, each representing an input node to a node in the hidden layer.

3. **CMA-ES based**: the method consists of using Covariance Matrix Adaptation - Evolution Strategy instead of Genetic Algorithms. However, the training of the neural network weights is no different from the previously explained *Genetic based*, where a vector of weights and biases is used to train the dataset. However, instead of using the GA, we use CMA-ES in order to evolve the weights to fit the training data, using the validation set RMSE to guide the fitness function. However, now instead of finding the best architecture with GA, we will also use CMA-ES.

3.2 Optimizing Architectures

– **Genetic Algorithm**: we use Gray binary encoding to define a genotype, which we will be translated to the phenotype according to a mapping function. To find the best architecture, a 7-bit binary string has been conceived for the genotype, where we will control three different parameters accordingly.

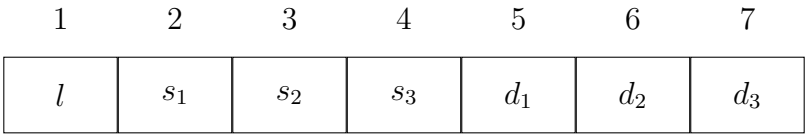

Figure 2: Definition of the chromosome in the genetic algorithm for the synthetic dataset.

where l defines if the output layer should be linear (1) or non-linear (0), the s represents the size parameter codified by three bits, thus, it can represent values up to 7. To set the size, it will be mapped as $size = 2^s$ thus taking the values $size \in [1, 2, 4, 8, 16, 32, 64, 128]$. Similarly, d can take the same values and is mapped as $decay = 10^{-(d+1)}$. This approach is similar to the one proposed in [1]. At each evolution, there are crossovers and mutations between populations, and the fitness function is computed as the RMSE of the architecture on the validation set. We have left the default parameters of cross-over and mutation that R comes with, and a population size of 50.

– **CMA-ES**: similarly, we use CMA-ES instead of Genetic Algorithms to also find the best architecture. However, the actual function needs to have numeric input instead of the encoding chromosome. Thus, we modify it to be continuous between the same minimum and maximum values. This is no problem for size and decay, however, the linear part will have to be removed since we cannot encode it correctly. This will also allow us to have a simpler search, and we will use the same result obtained from the GA as linear or not. Again, we train for each architecture a neural network and find the best architecture using the validation set. We have done the number of offsprings to be 50, with an initial step of 1.5.

Once the architectures have been selected for each type of algorithm, we train the neural network for more epochs – or generations – and obtain the unbiased estimate of the RMSE on the test data partition, which we have not used until now. Additionally, the elapsed total time of training is also measured, to have an estimate of the training complexity of each framework.

4 Results

For each of the different methods, we try different combinations of hardness and noisiness of the data. The following presented results are the average of the 3 different runs. The runs were performed with the seed removed and manually changing the hardness and nosiness of the data generation. Each time, the Rmarkdown was run 3 times and the results were written manually in a spreadsheet. Our initial intent was to write everything down in the Rmarkdown file, however, due to the complexity of each run, this was omitted in favor of "manually" generating the results file. The current Rmarkdown file returned

has a seed to replicate the results once, however, if you want to do the average, you should remove or change it at every run.

Hence, for different combinations of hardness and noisiness, we will run the different frameworks, and return the final average output of each dataset. We only report the final averages, the fitness function at each generation for the architectures, Figure 3, are omitted since they do not bring that much information and are dependent on each run.

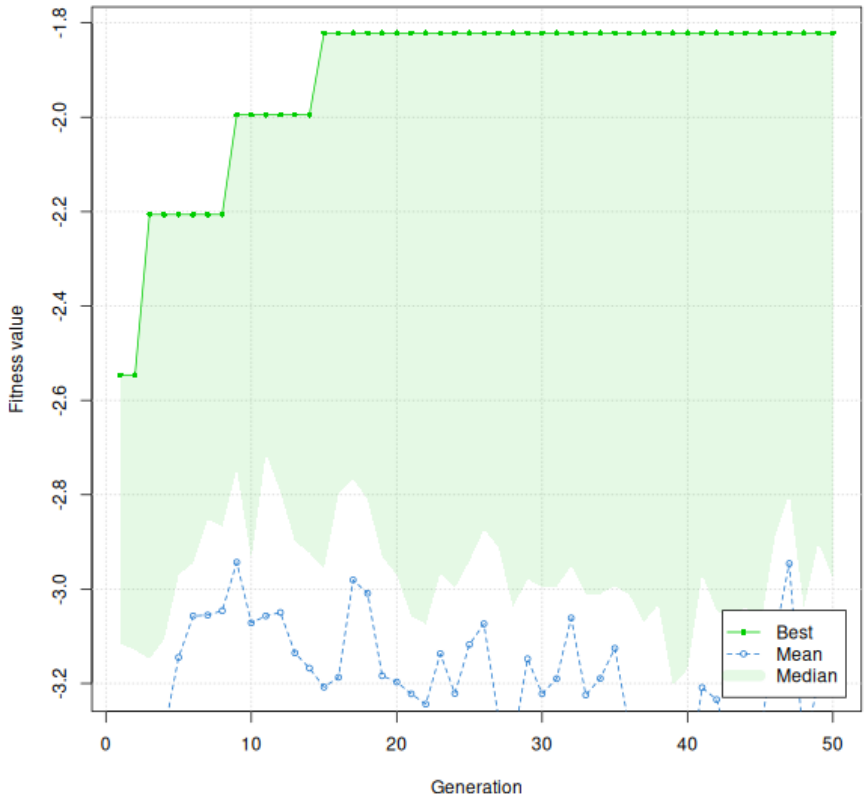

Figure 3: Fitness function evolution at each generation, for the search of the best architecture using backpropagation for training the neural net.

The combinations tested are:

Case 1) Hardness = 1 and noisiness = 1. Thus, a correlated problem where the data is not very noisy.

Case 2) Hardness = 1 and noisiness = 5. Thus, a correlated problem where the data is quite noisy.

Case 3) Hardness = 0 and noisiness = 1. Thus, an uncorrelated problem where the data is not very noisy.

Case 4) Hardness = 0 and noisiness = 5. Thus, an uncorrelated problem where the data is quite noisy.

4.1 Case 1

Algorithm	Linear	Size	Decay	Val Err.	Test Err.	Time (s)
Derivative	False	128	0.0010000	0.6445836	0.8918046	140.218
Genetic	True	1	0.1000000	0.7604663	2.6670950	375.436
Evolutionary	True	110	0.0038769	0.6593117	2.3304693	1071.339

Table 2: Case 1) Hardness = 1 and noisiness = 1 average results. For the parameters, we took the most frequent value.

Overall, we see that the derivative method results in the best validation error, thus, if this was a real world case, we would use the Derivative based neural networks as the best model. The evolutionary CMA-ES framework obtained a similar Validation error, but still worse than the derivative. Additionally, both these models resulted in very similar network sizes and decays, but the elapsed time of finding the best architecture and training resulted much worse in the Evolutionary setting.

Regarding the Genetic based, it only resulted in a size of 1 as the best model, seeming a clear outlier than the other two models. Moreover, it returns worse validation error, and the elapsed time is worse than the derivative, but not as bad as the evolutionary. As for the linear output, we can see that the genetic and evolutionary returned the same value, while the derivative said to not use a linear output.

In conclusion, for this dataset, we should go with the derivative method, as it resulted in the lowest RMSE in validation. Thus, a network with non-linear output, 128 nodes in the hidden layer, and decay of 0.001. This results in an unbiased estimated error of the RMSE as 0.8918046.

For the proceeding cases, we will quickly comment on this and compare the impact that the hardness and noisiness have, if any. The comments pertaining to the elapsed time are the same for all the cases.

4.2 Case 2

Algorithm	Linear	Size	Decay	Val Err.	Test Err.	Time (s)
Derivative	False	64	0.0100000	1.262427	2.603893	133.280
Genetic	False	4	0.1000000	0.7106575	2.127106	311.154
Evolutionary	False	115	0.0002091	1.368029	6.005472	1015.851

Table 3: Case 2) Hardness = 1 and noisiness = 5 average results. For the parameters, we took the most frequent value.

All cases returned nonlinear output, and regarding the validation error, we would go with the Genetic since it is the lowest. Again, the derivative and evolutionary cases seem pretty similar between themselves, with more or less the same validation error, but are pretty different in number of neurons in the hidden layer as well as the decay, moreover, they are very different from the genetic in regard to the number of neurons. Thus, the model we would select for this dataset, would be the Genetic framework with nonlinear

output with 4 nodes in the hidden layer and a decay of 0.1, resulting in a unbiased RMSE of 2.127106. Finally, there seems to be a bit of overfitting, since the test error is returning almost double the validation error.

Overall, increasing the noisiness seems to have produced more or less the same best architectures. The amount of nodes has been reduced a lot in the derivative methods, but the genetic have almost the same. Moreover, we can see that the decay has increased as well for the derivative method, while in the evolutionary it has decreased. The elapsed time seems pretty much equivalent, which is to be expected since we have not really modified any parameter that directly modifies these. Thus, all the algorithms are doing the same amount of iterations and generations, in both noise levels. We can also observe how, since the noisiness increased, the RMSE is larger for all the frameworks, as it is having a harder time "catching" this intrinsic noise, this is true for all frameworks except the genetic where the error decreased.

4.3 Case 3

Algorithm	**Linear**	**Size**	**Decay**	**Val Err.**	**Test Err.**	**Time** (s)
Derivative	False	2	0.0001000	0.5129859	1.7454011	98.688
Genetic	False	4	0.0100000	0.4759518	3.2792682	371.649
Evolutionary	False	108	0.0462164	1.0493156	4.3409770	1231.424

Table 4: Case 3) Hardness = 0 and noisiness = 1 average results. For the parameters, we took the most frequent value.

Surprisingly, the validation error is very small for all cases, but the genetic results in the smallest, so we should use it. The biggest difference we see now is that the derivative method also returned in a very small network, other than that similar comments from the previous cases can be said when selecting. We would select the Genetic, which results in a nonlinear output with size 4 and decay 0.01, resulting in a true error of 3.2792682. We should not do this, but checking the others test results, we can see how choosing the derivative would have resulted in a much better network that does not overfit as much.

Comparing these with the hardness at 1 and noisiness at 1 from case 1), we can see that the test errors have degenerated much more than before. It seems that increasing the hardness while maintaining the same level of noise has made the problem more difficult to generalize, resulting in this degeneration observed in the test set. Remarkably, we see that the derivative method has reduced by a lot the number of nodes in the hidden layer and decreased the decay quite a lot, resulting in a time save as well.

4.4 Case 4

Algorithm	Linear	Size	Decay	Val Err.	Test Err.	Time (s)
Derivative	True	128	0.0010000	1.9158700	4.330238	132.942
Genetic	False	16	0.0100000	2.0775559	4.374379	491.645
Evolutionary	False	79	0.0477272	2.4542691	6.679006	1194.108

Table 5: Case 4) Hardness = 0 and noisiness = 5 average results. For the parameters, we took the most frequent value.

The best validation error comes from the Derivative method, thus, we should use this. Comparing the methods, we see that the genetic returns more than it has ever returned for size, but is still the method that uses the least amount of neurons. The decays also seem pretty similar, especially between the genetic algorithms, however, there is a difference of one order of magnitude for the derivative method. Anyway, using the derivative method, we would use 128 nodes with a linear output and a decay of 0.001, resulting in an unbiased estimation of the RMSE of 4.330238, which seems to overfit as well.

Comparing it to case 2), which is with Hardness = 1, we see that most of the frameworks decided increasing the number of hidden layers. Additionally, the increase of hardness also resulted in a degeneration of the overall errors. Comparing it to case 3), which has the same hardness, but less nosiness, we can also see that the increase of noisiness affected the networks in selecting larger architectures at the cost of overfitting, and worse error overall.

4.5 Comments

Overall, we see that increasing the hardness of the problem decreases the ability of the neural networks to generalize and end up overfitting more. The same comment can be said about the noisiness. Nevertheless, the overfitting is perhaps due to the implementation, as once we found the best architecture, we did one final fit of the neural networks with an increased number of iterations/generations/population size. Additionally, the resulting elapsed times depends more on the number of iterations, population size, and other parameters of the evolutionary algorithms than the problem hardness or noise. Finally, on each of the cases, the resulting architectures are very different depending on the framework used to train the networks. This was to be expected, since a great deal of the performance of the network depends on the architecture, and the learning will be different for each algorithm. However, even though most architectures were different, the resulting performances did not differ as much, referring to the fact that the error landscape is multimodal since different architectures may have similar performance.

Regarding the comparisons between algorithms, we have already done it at each case. However, overall, it seems that the derivative method works the best in most cases. Even though sometimes we selected the Genetic, if we "cheat" and look at the test errors, we see how in all cases, expect case 2), the resulting error was the least when using the derivative method. Thus, it seems that using the derivative method, at least when we have a differentiable and continuous function, reigns king. Comparing the evolution strategies between themselves, we can see that the Genetic works better than the CMA-ES in all cases. Only in case 1) did we find a case where the CMA-ES returned better

error than the GA. After reviewing the literature that was shared to us for the completion of this project, this fact seems more obvious, as most of it was pertaining GA and only few mentioned CMA-ES.

Additionally, CMA-ES resulted in worse elapsed time than the GA and derivative. However, we should take the comments about the elapsed time with a grain of salt, as we are doing more iterations and generations on the CMA-ES than both others. So while we could be quick to dismiss it due to its complexity, it can be the case that with different parameters we could achieve similar times. Additionally, we have not tested the case of training the neural network with CMA-ES and find the best parameters with the GA, which could perhaps improve the selection and speed-up the framework, resulting in not so bad times.

Moreover, we cannot dismiss the evolutionary algorithms so quickly, as it could be that with more iterations, different populations sizes, etc. could achieve better results than the derivative method.

5 Conclusion

After fulfilling the practical, having analyzed in depth the generation of synthetic data and implemented different frameworks for the use of Evolutionary Algorithms for Neural Networks, we can draw the following conclusions:

- We have seen that setting the architecture of a neural network is not a trivial task, and changing the different components will result in different performances.
- We have adequately learned a different technique to learn neural networks without the need of backpropagation. This, in most cases, has resulted in worse performance than using the typical backpropagation algorithm, however, it does hold some advantages over backpropagation, such that it could have non-differentiable or non-continuous error functions. As well as having discrete information about the network, which would help in reducing the size of the architecture. Unfortunately, in our toy case, these benefits were overshadowed by just using a classic error function such as the RMSE.
- We have tested a different technique to optimize the architecture of the network. While this could usually be done through cross-validated grid search, we have achieved good results using GA and CMA-ES to optimize the size and decay of the network. Moreover, it is quite possible that we have explored the space more efficiently than using a grid-search, thanks to the selection of Evolutionary Algorithms. We cannot confirm this with adequate testing, though.
- The elapsed time is greatly dependent on the amount of iterations, population size, etc. the Evolutionary Algorithms are required to do. However, keeping the parameters more or less equal, we have seen that the evolutionary techniques take more time on average than the derivative techniques to train the neural network to achieve the same performance. These results are not conclusive and more formal testing should be done.

Nevertheless, there are some drawbacks in our analysis, which should be further formalized or taken into consideration for future work:

- The amount of time was limited, thus, the number of iterations the GA and CMA-ES were allowed to do was greatly cut. Additionally, the population sizes and number of offspring was also limited. In most cases, we observed that the fitness function plateaued for the architecture, however, this was not always the case for the training of the neural net with the evolutionary techniques. Thus, it could be the case that increasing these parameters of the Evolutionary Algorithms would increase the resulting performance.

- Statistical comparisons should be done to analyze the differences between groups, for example, either a one-way ANOVA to test differences between group and a post-hoc Tukey-HSD test, or a non-parametric Friedman with its corresponding post-hoc test. This would allow us to truly compare if the differences between the frameworks is statistically significative or not. Thus, allowing us to actually select which is the best framework for the presented problem, but this is way over the required work of this project.

- The architecture and training of the network could be done in a more accurate way by using nested cross-validation to estimate the error at each fitted architecture. Otherwise, we are risking the possibility of over-optimistic estimations of the error.

- The specific parameters of the evolutionary algorithms have not been tested much. After a brief initial testing, we decided to leave them at R's defaults. Thus, it could be the case that modifying these values could improve convergence and performance of the algorithms, however, modifying them would have increased drastically the complexity of the already complex project.

- Finally, the generated problem is perhaps too difficult and other problems should be studied. We got sunk into the sunk-cost fallacy, and instead of switching to one of the problems from `mlbnech`, we decided to complicate it, to such a degree that it is difficult to estimate the error. The use of a multivariant normal distribution, while useful to create correlated variables, made it non-trivial to estimate the error without removing the randomness. Thus, for future studies, perhaps it would be easier to tackle a simpler problem instead of this monstrosity that we created. Nevertheless, it was a learning experience to see why most benchmarks are well-defined and are widely used, instead of randomly creating a new synthetic benchmark, which is a difficult task in itself.

6 References

[1] Alejandro Baldominos, Yago Saez, and Pedro Isasi. "Evolutionary convolutional neural networks: An application to handwriting recognition". In: *Neurocomputing* 283 (2018), pp. 38–52.

[2] David J Montana, Lawrence Davis, et al. "Training feedforward neural networks using genetic algorithms." In: *IJCAI*. Vol. 89. 1989, pp. 762–767.