University of Minho
School of Engineering

Inês Alves - A81368, Pedro Pinto - A80741, Marta Ribeiro - A82474

Multi-objective evolutionary approach to Artificial Neural Networks architectural design in breast cancer detection

Abstract

In this work, we study a multi-objective evolutionary approach to the design of multi-layer perceptron architectures. In a case study where the purpose of the classifier is to detect cancer in records from mammography exams, we find that multi-objective genetic algorithms provide a better way of optimizing our prediction models to our real needs, when compared to more human-centered approaches, upon which we establish a baseline.

The models that result from the evolutionary process reveal a dramatically increased recall metric (true positive rate), while maintaining (and even slightly improving) accuracy.

Keywords: Artificial Neutal Networks, Genetic Algorithms, Evolutionary Algorithms, Breast Cancer, Data Science, Machine Learning, Deep Learning.

April 27, 2020

Contents

1 Introduction

The use of machine learning (especially deep learning) has been increasing at a rapid pace in many areas, and it's potential to contribute to medicine has been widely demonstrated.[6]

The expertise required to produce high performing machine learning models, however, are not easy to attain and have revealed themselves to be in a rapidly growing demand. To mitigate the disparity between supply and demand of these expertise, there have been efforts to automate parts or the entirety of common data science pipelines. The most common involve the implementation of AutoML software that attempts to limit the user's interaction to the gathering of the data and the specification of the target variable which one desires to accurately predict.

In this work, we explore a more conservative approach, which consists in limiting the scope of the automation to the optimization of the learning models, while making use of **evolutionary genetic algorithms** to select the best configurations for a particular type of **artificial neural networks**.

Despite neural networks admittedly not being the best solution for every problem (especially less complex ones), we use them because of their increasing importance in science, and since their configuration is rarely a straightforward process, we hope to be able to extrapolate the conclusions of this work into more complex situations.

2 Data

The dataset we will be using in this work is consists in the records of 961 mammograms. For each of these records, we have six attributes:

- BIRADS - refers to the mammography assessment categories. These are standardized numerical codes that allows concise and unambiguous understanding of patient records (1-5).
- Age - corresponds to the age of the patient.
- mass-shape - corresponds to the shape of the mass found (round=1, oval=2, lobular=3, irregular=4).
- mass-margin - corresponds to the margin of the mass found (circumscribed=1, micro-lobed=2, obscured=3, ill-defined=4, spiked=5).
- mass-density - corresponds to the density of the mass found (high=1, medium=2, low=3, presents fat=4).
- cancer - 0 if it is benign and 1 if it is malign.

2.1 Data Preparation

We start this process by loading the data to a dataframe object. After that we analyse the data and found that there were null values. We check that there were values misintroduced too, like a BIDRADS value of 6, for example. To later deal with this problem, we marked all of these occurewnces with NaN values.

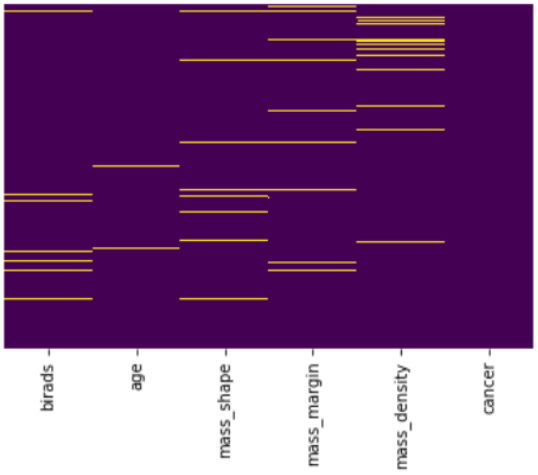


Figure 1: Null and misintroduced values in the dataset.

At first, we removed all of the in the majority class that featured these situations. In order to impute the remaining missing values, we used a **k-nearest neighbours (KNN) imputation algorithm**. In this approach we won't

use more balance techniques, because the undersampling used before had left the dataset completely balanced.

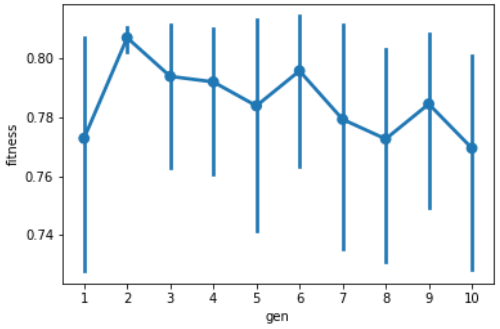


Figure 2: Preliminary analysis of the evolution with the KNN + Undersampling strategy of data preparation

The results weren't what we expect, so we decided to try another solution. Again, we converted all misintroduced values into NaN and used KNN to impute the missing values. The difference was that in this new strategy, we used the **SMOTE[3] over-sampling** strategy in order to balance the dataset.

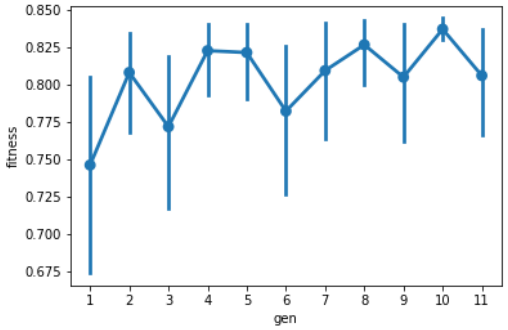


Figure 3: Preliminary model fitness evolution using KNN + SMOTE strategy

Based on the results presented for each of the approaches we decide to use the latter.

We also **scaled the data** so that it would be more adequate for use in neural networks, and **one-hot encoded all of the categorical features**, so that possible underlying non-linear relationships between features wouldn't be lost. Since we one-hot encoded the categorical features, the only column that was scaled was the age.

Finally, we dropped the BIRADS column, since one of the challenges in this work was not to use it (since it results from a human classification based on the other features). We didn't drop that column before since it was useful for the imputation and balancing processes.

3 Learning Model

Implementation and experimentation with different artificial neural network (henceforth simply referred to as neural networks) architectures has been made easy in the last years with the emergence of machine learning frameworks such as Tensorflow[2] and Keras[4], but the optimization of the networks' structure and hyper-parameters is rarely a straightforward process.

There are several types of neural networks , serving different purposes and arising from various topologies (layered structures, most commonly) and interactions (such as the so called activation functions) between their elementary units, the **artificial neurons**.

The type of architecture generally used in classification problems such as our case study is the **multilayer perceptron (MLP)**. The formulation of our problem also suggests taking some decisions such as using a single neuron as output layer and **binary crossentropy as loss function** for our model. The input layer's structure is determined by the shape of the data after preprocessing.

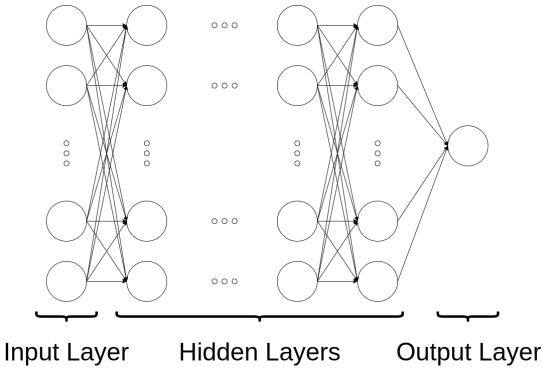


Figure 4: General representation of a typical MLP binary classifier.

3.1 Evolved Model Features

To accelerate the evolutionary process (and reduce the chance of it randomly selecting bad solutions) we established some ground rules to limit our possible model configurations.

- Between the input and output of the network there is an **arbitrary number of layer blocks**
- A block is composed by a **dense layer** (fully connected), a **dropout layer** (randomly ignores connections with the previous layer, which sometimes helps preventing over-fitting) and a **normalization layer**
- Each of the layers have characteristics that are subjected to evolutionary selection (**size and activation function** in the dense layers, dropout ratio in the dropout layers)
- The model's **learning rate, batch size, and classification threshold** are also subject to evolution

3.2 Baseline

In order to compare the evolutionary approach following out genetic algorithm with a more rudimentary (and common) manual approach, we set up an MLP with an architecture based on our intuition and manual trial and error (which is the method followed in many situations).

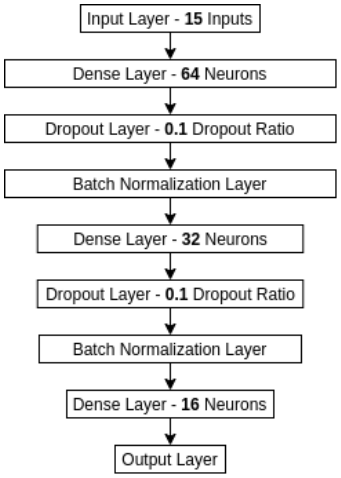


Figure 5: Architecture of our baseline model. We adopted a learning rate of 0.001, batch size of 32 and a classification threshold of 0.5.

3.3 Tools and Implementation

In order to implement our MLPs we used Tensorflow and Keras, which provide a vast set of layer types, metrics and utility functions (with comprehensive documentation).

In order and evaluate the performance of our baseline and final models, we used the Tensorboard functionality.

4 Evolutionary Algorithm

The implementation of most evolutionary algorithms can be generally described as the execution of the following steps:

1. Generate a **population**
2. Until a stopping criterion is met, repeat the following steps:
   (a) Evaluate the **fitness** (or quality) of each individual
   (b) Select a subset of individuals from the population

(c) Generate new individuals (offspring) through processes such as:

- **Mutation**
- **Crossover** (or mating)

(d) Replace a subset of the population with the new individuals

But before diving into the criteria by which any of these steps take place, we need to define the object of evolution itself: genetic code (the algorithm's "DNA").

4.1 Genetic Code

With efficiency in mind, we represented the genetic code of a model (henceforth we interchangeably use the terms model and individual) as an array of 16-bit floating point numbers.

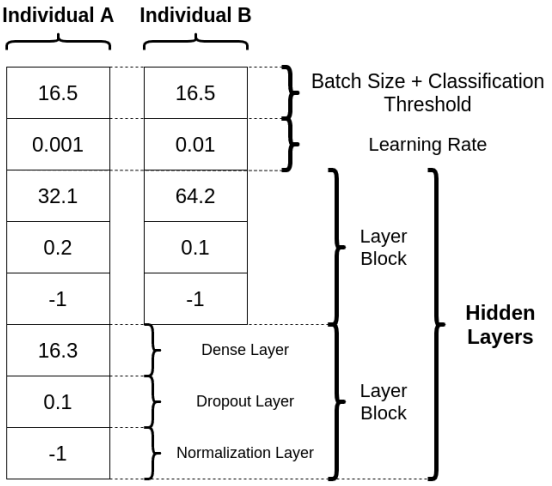


Figure 6: Representation of two possible individuals. The integer part of the first element of the array encodes the batch size, while the decimal part is the classification threshold. The second element encodes the learning rate of the model. The remaining elements encode hidden layers. Dense layers are represented by numbers greater than 1, where the integer part is the number of neurons and the decimal part encodes the activation function (mapped in a table). Dropout layers correspond to numbers between 0 and 1, and those numbers are the dropout rates. Normalization layers are encoded as negative numbers (fixed at -1). Although we allow individual layers to suffer undergo a process of mutation, we force the neural network architecture to be organized in blocks. These blocks can be duplicated or removed from the network, that being the only way of adding or removing layers.

4.2 Initial Population

The characteristics of the individuals in the first generation must be well chosen, in order to **maximize exploration** while also allowing the algorithm to **converge conveniently faster**.

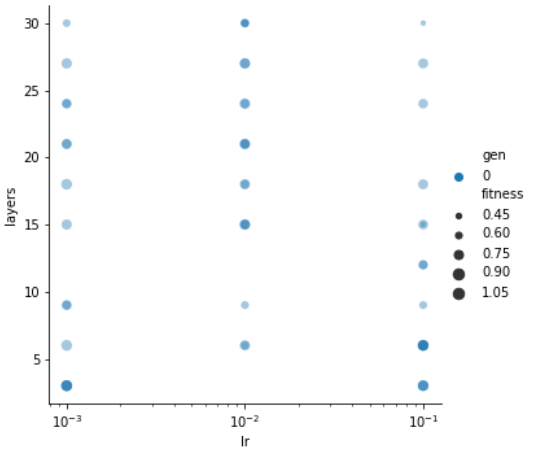


Figure 7: Initial number of layers and learning rates

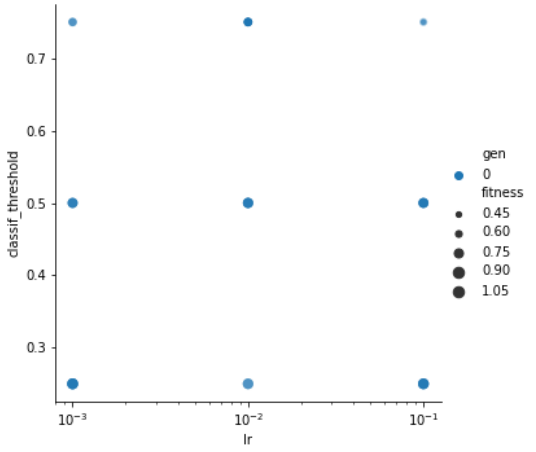


Figure 8: Initial values for classification threshold and learning rate

Each single characteristic that is subject to evolution is distributed uniformly in respective set of initial values:

- Batch Size - {8, 16, 32}

- Learning rate - {0.1, 0.01, 0.001}
- Number of Hidden Layers -{3, 6, 9, 12, 15, 18, 21, 24, 27, 30}
- Classification Threshold - {0.25, 0.5, 0.75}
- Layer Activation Function - {Rectified Linear ('relu'), Sigmoid, Hyperbolic Tangent ('tanh'), Linear}
- Number of Neurons (dense layers) - {16, 32, 64}
- Dropout Rate (dropout layers) - {0, 0.05, 0.1}

4.3 Evaluation

The evaluation method is the main driving force of any evolutionary algorithm, since it provides the criteria based on which individuals that better suit our needs are selected.

In order to implement a successful algorithm, the evaluation function must be descriptive of the success of a given individual at solving the problem at hand.

In that sense, we came up with our two main **objectives** for this case study (in descending order of importance):

1. Save lives (incorrectly classify as few as possible possible cases, in other words, reduce the false negative rate of our model as much as possible).
2. Correctly discard a conveniently high percentage of negative cases, thus saving time and money to the healthcare system.

Considering these objectives, we can already infer that metrics such as **accuracy or cross-entropy would not be successful evaluation functions** in our genetic algorithm. In order to enforce our individuals' capability of successfully detecting positive cases, we used the **recall** (also known as true positive rate, or sensibility).

If we only considered the recall, however, we could be selecting models that always gave a positive prediction. In this sense, we also consider another the **AUC** (are under the ROC curve) metric of the models, which translates our secondary goal of creating models that are better at distinguishing positive and negative cases (thus raising less false alarms).

A rudimentary approach to the usage of these metrics would be calculating a fitness based on a weighted sum, thus adopting a single-objective situation. A strategy of this sort would be easy to implement and understand, however it wouldn't be as successful as using a **multi-objective approach**.

$$\vec{F}(\vec{ind}) = \left\langle Recall_{tr(\vec{ind})},\ AUC \right\rangle \iff$$

$$\vec{F}(\vec{ind}) = \left\langle TPR_{tr(\vec{ind})},\ \int_{t=0}^{1} TPR_t dt \right\rangle$$

Figure 9: Our evaluation function returns a vector with the recall and the AUC metrics. If we further explore the formulae for these metrics, we can understand that both are highly dependant on the true positive rate (TPR). Despite the recall only operates on the classification threshold of the evaluated individual, the AUC is calculated with considering the whole range of possible thresholds $[0, 1]$. Ultimately, this allows us not only to select the thresholds that yield the best results, but also to consider individuals that might perform better when their classification threshold changes.

4.4 Selection

The selection of the subset of individuals that get the chance to reproduce and mutate, originating the next generation, is also a crucial part of any evolutionary optimization approach.

This part of the algorithm is tightly related with the evaluation function, and as we've stated before, we have taken a multi-objective approach to this process.

Our selection strategy is the same as the one used in a popular algorithm known as **NSGA-II[5] (Non-dominated Sorting Genetic Algorithm)**, and consists in two key concepts:

- **Non-dominant sorting** of the populations, which consists in determining the Pareto front [1] to which each individual belongs (by comparing it to all the other

individuals), and including the best performing fronts in the next generation.

- **Crowding distance** is used to select individuals from the last chosen front (which probably won't fit in the remaining population size). This results in the most extreme individuals in this front being selected, thus encouraging exploration.

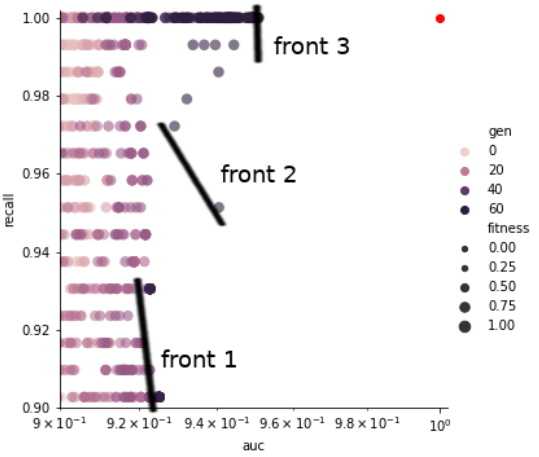


Figure 10: Several possibilities of what might have been parts of Pareto fronts in our evolutionary process. The most interesting is "Front 1", since it features two separate individuals that survived through many generations, until eventually there was a breakthrough that lent their fitness obsolete. The red dot is a hypothetical ideal model.

4.5 Crossover

Crossover (also referred to as mating) allows us to combine characteristics between two individuals.

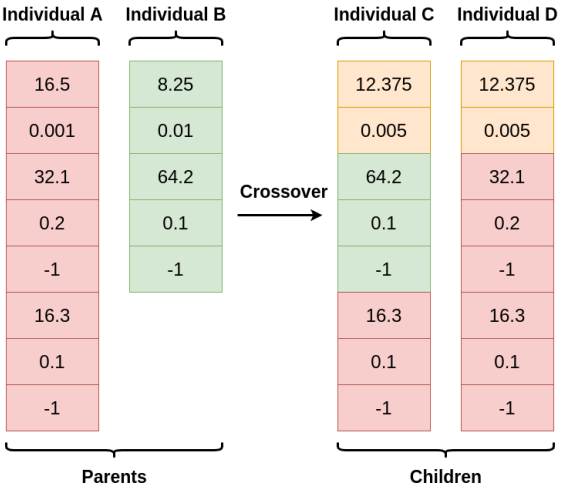


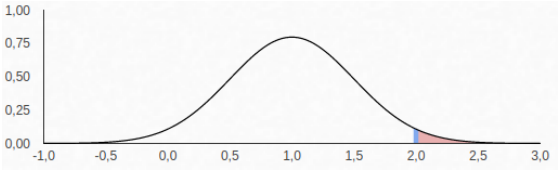
Figure 11: Representation of a possible crossover scenario, where two individuals (even with different numbers of layer sizes) originate two children with randomly selected characteristics present in their parents. The batch size, learning rate and classification threshold of the children is the average of their parents'. Each block of the a children's architecture is randomly selected between the corresponding block in each of the parents.

Our implementation of this process allows that even **models with very different architectures can originate intermediate solutions** that might represent breakthroughs in the evolutionary process.

4.6 Mutation

The mutation process allows us to originate new individuals from slight variations in characteristics from existing ones. Our mutation strategy consists in defining a probability of each characteristic of an individual to undergo mutation. When mutation happens for a characteristic (as if it was a gene), a new value is generated according to a specific rule.

In most of the characteristics, we multiply the old value by a random variation factor that follows a **normal distribution**. The only exception is the activation functions, which (being a categorical feature) are chosen in a uniform distribution between the available values.



(a) Normal distribution, or $\mu = 1, \sigma = 0.5$. The probability of the variation being 2 or higher (making a given value double or more) is of 2.275%. We make sure that the variation doesn't assume negative values.

$$NewValue = OldValue \times X, \qquad X \sim \mathcal{N}(1, \sigma)$$

(b) Formula for calculating the new values the characteristics of an individual, when mutation occurs. The mean value of the distribution is always 1 (originating a null variation), and the standard deviation is chosen according to how drastic we want the variations to be (in most cases, we use $\sigma = 0.5$).

Figure 12: Mutation of characteristic values

In our implementation of the mutation process, it is also possible for a **block of layers in an individual to be doubled or removed** from the architecture.

Figure 13: Example of two possible mutants originated from an individual. Notice that some values are slighly changed, and that in individual X the second block of hidden layers was duplicated whilst in individual Y the first block was removed.

4.7 Tools and Implementation

To develop our evolutionary algorithm, we used the **DEAP[7] framework**, which provides some very useful function wrappers and even full algorithm components that facilitate immensely the work in projects such as this one.

In order to make the algorithm more **efficient**, we defined the genetic codes as NumPy[8] arrays of 16-bit floating point numbers.

5 Experiment

5.1 Experimental Process

To apply our evolutionary algorithm we split the dataset in three balanced (equal number of true and negative cases) parts:

- Train dataset - 81%
- Validation dataset - 9%
- Test dataset - 10%

It is important to point out that the test dataset **does not contain** synthetically generated values (**SMOTE**) nor any imputated value (**KNN Imputation**).

We devised some ground rules to avoid over-fitting:

1. Each individual (model) is trained in the training set, and it's fitness is determined by metrics measured on the validation set
2. Each individual is trained for 10 epochs, and the average of the highest 3 values of the objectives of the evolutionary process is returned by the evaluation function
3. The **test dataset is never used in the experiment** until we compared the baseline model to the models resulting from our evolutionary process
4. Our population has 50 individuals, and 10 of them are selected in each generation

We record every individual's architecture, hyper-parameters and evaluation scores in a

dataset that allows us to not just study the evolution of our experiment, but also restore any individual that ever existed in the experiment.

We ran our experiment for 50 generations.

5.2 Results

5.2.1 Evolutionary Progress

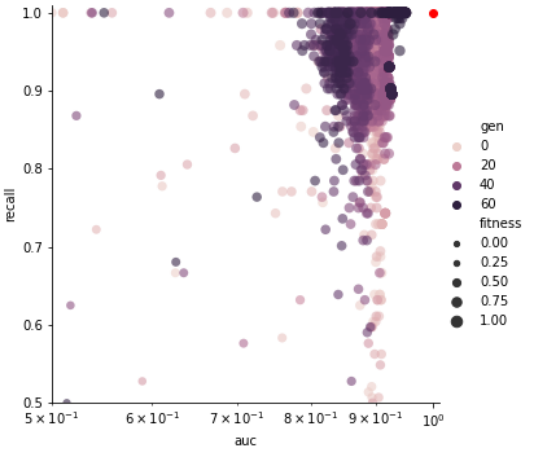


Figure 14: Evolution of the objectives, AUC and recall. The red dot is a hypothetical ideal model. We can see that the best solutions (darker, since they're the population of more advanced generations) are almost exclusively approaching the ideal scenario from the maximum recall value. This indicates that it is not only possible but actually easier to get better results with more sensible models (prone to make positive classifications).

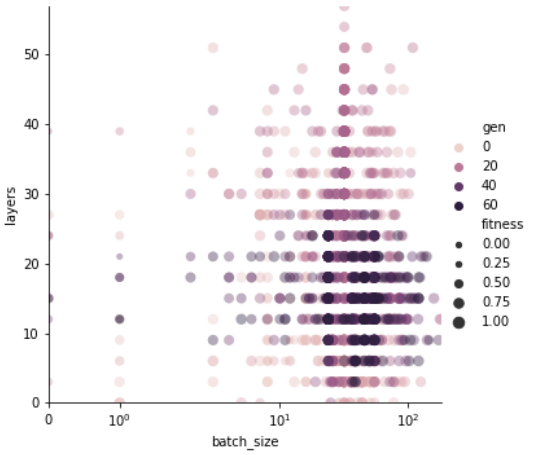


Figure 15: The number of layers converges to values between 9 and 24, while the more common batch sizes are between about 24 and 64

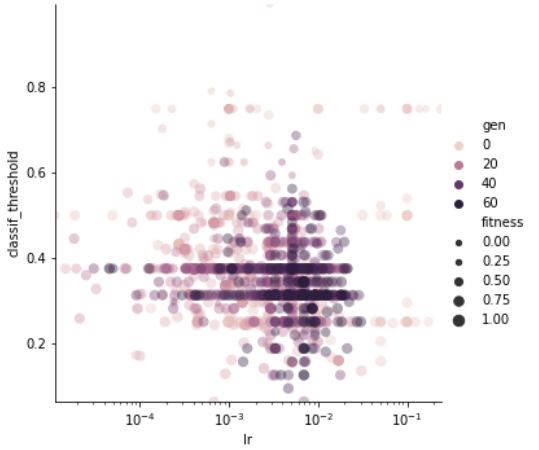


Figure 16: The classification threshold converges to values between 0.30 and 0.35, which is coherent with the fact that the recall is the main objective in our evolutionary process. The learning rate converges to values around 0.01.

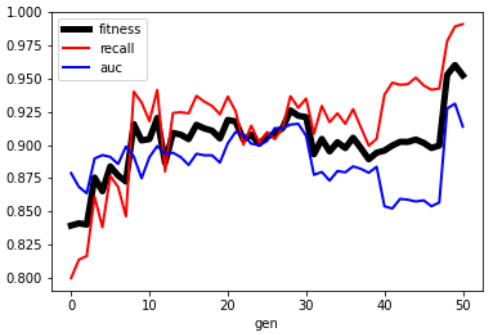


Figure 17: Evolution of the average of the evolution objectives through the generations. The black line is the average between the two metrics.

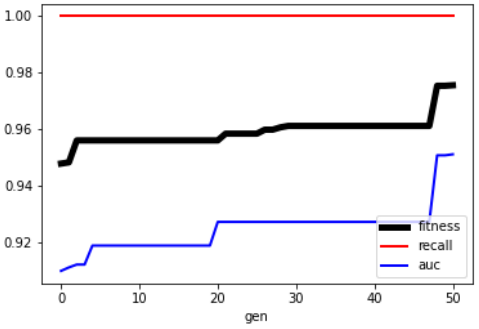


Figure 18: Evolution of the maximum value of each evolution objective through the generations. The black line is the average between the two metrics.

5.2.2 Performance Improvements

Finally, we selected three individuals from the whole evolutionary process:

- Highest recall
- Highest AUC
- Highest fitness (average between the two objectives)

We compared the selected models to the baseline previously set up.

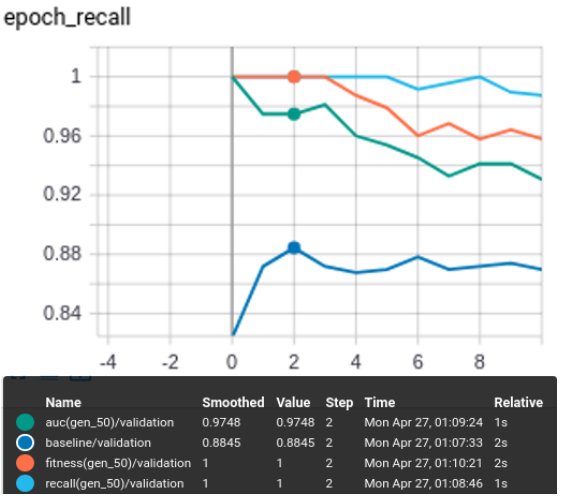


Figure 19: The models resulting from our evolutionary process reveal massive improvement in the recall metric, which reflects a high value to our problem, as previously mentioned.

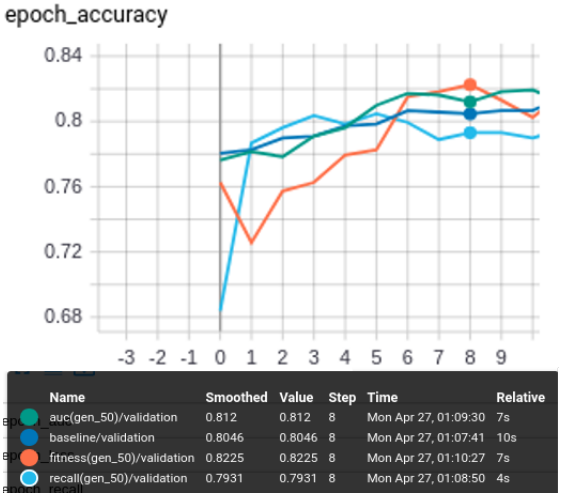


Figure 20: The more conservatively selected models reveal a better accuracy than the model with maximum recall, as expected.

The baseline model never performs better (according to any metric) than the models selected by our genetic algorithm.

5.3 Discussion

The first observation we can make is that, although in the first generations there is cer-

tainly a more gradual progress of the population's fitness, **in more advanced generations that evolution is the result of rare evolutionary breakthroughs** introduced by mutation and crossover.

This makes the creation of **stopping criteria** more difficult, since breakthroughs might be separated by many generations (just as we observed a hiatus between about generations 29 and 48), giving a false impression of convergence. In slow evolutionary processes (as in this case), we feel that the best practice is to either manually and carefully analyse the changes occurring in each generation, or to implement more sophisticated stopping criteria that take into account if the multi-dimensional space of possible individuals around each individual of the population has been explored (thus proving that a breakthrough would be highly unlikely).

The evolutionary process successfully selected models that excel at detecting positive cases without losing accuracy. Furthermore, it is noteworthy that even though that increasing accuracy was not the purpose of our genetic algorithm, the baseline model performed worse than the others in this metric. These results are very positive.

6 Conclusion

We conclude that evolutionary algorithms are a **valid tool** to use in neural networks architectural design, given that they facilitate the modeling of real world use cases.

The **methodical exploration and selection** of the fittest variants of a given general architecture surpasses our human capabilities to remember a few characteristics that might help solving our problems, based on experience.

The **computational cost** of using evolutionary approaches, however, is extremely high and it is only a wise choice in situations where we can potentially save lives, as in our case study.

Bibliography

[1] *Pareto Efficiency*. Wikimedia, 2020.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[3] Kevin W. Bowyer, Nitesh V. Chawla, Lawrence O. Hall, and W. Philip Kegelmeyer. SMOTE: synthetic minority over-sampling technique. *CoRR*, abs/1106.1813, 2011.

[4] François Chollet et al. Keras. https://keras.io, 2015.

[5] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[6] Rahul C. Deo. Machine learning in medicine. *Circulation*, 132(6), 2015.

[7] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, jul 2012.

[8] Travis Oliphant. NumPy: A guide to NumPy. USA: Trelgol Publishing, 2006–. [Online; accessed ¡today¿].