Natural Computing Coursework 2

Abstract

Neural networks have become an important tool nowadays being useful in many domains including vision, games, language processing. However, the neural network based models can get very complex for complicated tasks and finding an optimal architecture is still a hard challenge and an open research field. In this paper, we explore the benefits of a genetic approach to solve this task for the known two-spiral dataset. Moreover, we also test the effectiveness of a particle swarm optimizer to train a neural network compared to the usual SGD. Our results show that PSO can perform at the same level as SGD with careful initialization and GA and GP do find better results than a grid-search for hyper-parameter tuning.

1. Introduction

In this paper, we go over three different tasks that explore different areas of Natural Computing. All three tasks relate to the *Two Spirals* classification task (Lang & Witbrock, 1988) (Alvarez-Sanchez, 1999), in which we want to decide if a datapoint belongs to the first or the second spiral. Even though the dataset is small (200 points in a 2D space), this task is relatively complicated due to the shape of the data. The train/test ratio is 50%. We use the google playground javascript framework for our experiments.

In the first task, we optimised the weights of a neural network for this problem using PSO (particle swarm optimization) (Eberhart & Kennedy, 1995). In the second task we used a Genetic Algorithm to optimise the shape of a neural network. For the third task, we solved the problem using GP (genetic programming) and CGP (cartesian genetic programming).

2. Task 1 - Particle Swarm Optimisation

We first tried to establish baseline models for both **SGD** and **PSO** on a shallow neural network (NN) architecture. We used *grid search* trying multiple hyper-parameter choices especially for **SGD** seeking a solid baseline model. However, for **PSO** the search for good hyper-parameters was a little bit more straightforward as good values for ω and α were established in the literature (Binkley & Hagiwara, 2008) with varying swarm size.

To first establish the search space for our main problem, depending on the architecture of the neural network, we have a certain number of model parameters that we try to optimize comprising of weights and biases. These indicate the **dimensionality** of the problem. If we define the set of weights and biases $(w_0, w_1, w_2, ..., w_n, b_0, b_1, ..., b_m)$, then a solution would be a tuple of length $m + n$, $(w_0, w_1, w_2, ..., w_n, b_0, b_1, ..., b_m)$ for which the objective function is minimized as much as possible. For **SGD**, we will optimize these model parameters with *Gradient Descent* and for **PSO**, we will denote a population - potential solutions - and we will do the minimization according to the well-known algorithm of swarm intelligence.

2.1. Baseline experiments with SGD

Our experiments for baseline models were conducted minimizing the mean square error on training dataset. Shallow designs with 4, 6 and 8 neurons were tried (corresponding to a search space of 24-48 dimensions) together with learning rates of 1, 0.1 and 0.01. For the activation function, we tested *ReLu*, *Sigmoid*, *Sin* and *RBF*.

ReLU Activation Function		
Neurons	Training Loss	Test Loss
4	0.447	0.505
6	0.363	0.508
8	0.342	0.492

Sigmoid Activation Function		
Neurons	Training Loss	Test Loss
4	0.398	0.516
6	0.449	0.471
8	0.439	0.491

Sin Activation Function		
Neurons	Training Loss	Test Loss
4	0.434	0.446
6	0.350	0.390
8	0.425	0.454

RBF Activation Function		
Neurons	Training Loss	Test Loss
4	0.350	0.480
6	0.244	0.369
8	0.225	0.374

Table 1. Training and test loss for SGD (learning rate = 0.01) using different activation functions and 1 hidden layer.

The point of these experiments is to discover what combination of hyper-parameters is likely to be good in future experiments when we try to improve on these.

We stayed consistent with **5 epochs** during training for each our experiments (we observed during our initial testing that this is usually enough), although in special cases that we will mention, more epochs are needed to assume convergence. The results using the best learning rate are shown in table 1.

For **SGD** training, we found out that 6 neurons provide good enough trade-off between complexity and performance. A learning rate of 0.01 is perfect for this task as going higher did lead to unstable performance in our experiments (although it is totally possible to work for deeper models) and *RBF* provided the best performance for a one-layer neural network- obtaining a test loss of 0.369.

We expected *RBF* to do a really good job, more efficiently than the other activation layers as it generally provides the most non-linearity to the network. This is a drawback for other, more complex architectures, but for this task, it seems in general that more non-linearity leads to lower loss and, therefore, higher accuracy. That is also why *ReLU* seems to provide such a bad performance, though, as we will see in our next experiments, it does match *RBF* for more complex neural networks architectures. As a plus, *ReLU* is so much faster than the other activations which can make a difference for **PSO** training which is slower than **SGD**. We will restrain from using *sigmoid* or *tanh* or *sin* in future experiments, not because they won't work or won't provide good performance (they certainly can and it was proven that they can (Karlik & Olgac, 2011), (Neal, 1992)), but because of the usual problem with vanishing gradient with these activations (Nwankpa et al., 2018), plus using *sin* as activation is hard to train because the periodic nature that can give rise to a 'rippling' cost function with bad local minima, moreover the derivative of *sin* is 0 in many points ($sin(\pi/2 + k\pi) = 0, k \in \mathbf{Z}$), this is not a problem for **PSO** but it is for **SGD**.

Swarm Size	Training Loss	Test Loss
10	0.362	0.443
20	0.284	0.380
50	0.326	0.548
100	0.269	0.374

Table 2. Training and test loss for baseline PSO with RBF activation function and 1 hidden layer with 6 neurons ($\alpha = 2$, $\omega = 0.9$)

Hidden Layers	Training Loss	Test Loss
1	0.444	0.451
2	0.453	0.481
3	0.451	0.505

Table 3. Training and test loss for PSO ($swarm\_size = 100, \alpha = 2, \omega = 0.9$) with RBF activation function and different number of hidden layers with 6 neurons each.

2.2. Baseline experiments with PSO

2.2.1. Setup

The experiments with **SGD** were very valuable to establish a good starting point for **PSO** training. There are several things that can severely affect the performance for a **PSO** minimization task: the starting point, values for ω and α, the size of the swarm and the dimensionality of each particle.

For the following experiments, the train loss stayed at the basis of fitness function definition for PSO, that we are trying to minimize. Afterwords, we will conduct experiments on another fitness function designed by us. To assert the performance of a model, we favour the models which get the fitness function closer to 0.

2.2.2. Defining a good starting point for pso hyper-parameters. Experiments.

As a starting point, all the particles are initialized with random float numbers from a certain zero-centered interval. It was really easy to provide a good enough value for this range as empirically, the weights for the baseline neural networks trained with **SGD** were mostly in the interval $[-1, 1]$, with some exceptions. Therefore, we chose this range to initialize the swarn.

For the size of the swarm, we tried to search for a good trade-off between computational time and performance. We tried swarm size of 10, 20, 50 and 100 with the best NN structure obtained from the previous experiments (Table 2). We decided to continue with a swarm size of 100 as it is generally better in the long run and yields the best performance. Very few experiments were also conducted with 500 and 1000 individuals in population. However, they were discarded as being too slow for the scope of this study, providing similar performance with the other models.

For the dimensionality of each particle, we surprisingly found out that decreasing the complexity of the model, from 3 to 2 layers for example, helped with that and increased the performance of the **PSO** with identical other hyper-parameters. The very best models were still achieved with 2 hidden layers, but in some scenarios, even 1 hidden layer outperformed 2 hidden layers (Table 3).

With regards to α and ω, we have a hypothesis that we prove in this paper. It is no secret that a good deal of exploration is needed for this medium-sized problem. This can be achieved with an $|\omega|$ closer to 1 and with overshooting forces (Mitchell, 1998), (Holland, 1992). However, setting ω close to 0 and simultaneously trying high values for α (4 or 5) would be too much and will lead to too much instability and the algorithm will probably not converge in 5, nor 10 epochs. **We empirically show that high values for ω and relatively low values for α outperform high values for ω and high values for α.** Moreover, we show that low values for ω and high values for α (4 or 5) would lead to similar performance. Table 4 is proof of our claims. The results are not surprising as several other studies were conducted on this matter (Kennedy & Eberhart, 2002) (Binkley & Hagiwara, 2008). We run these experiments multiple times to be sure that the results are consistent. From now on, we choose $\omega = 0.1$ and $\alpha = 4$ as it leads to the best results overall (Table 4). It seems that overshooting forces - swarm intelligence is more important in this task than one's personal best.

ω	α	Training Loss	Test Loss
0.9	4.0	0.432	0.468
0.9	2.0	0.233	0.340
0.1	4.0	0.152	0.262
0.1	2.0	0.472	0.481

Table 4. Training and test loss for PSO ($swarm\_size = 100$) with RBF activation function and 2 hidden layers with 6 neurons each.

SGD			
Inputs	Training Loss	Test Loss	Accuracy
Linear	0.042	0.178	87.6%
Non-linear	0.002	0.025	98.8%
PSO			
Inputs	Training Loss	Test Loss	Accuracy
Linear	0.152	0.262	83.6%
Non-linear	0.016	0.069	96.4%

Table 5. Comparing the performance for SGD and PSO with RBF activation function and 2 hidden layers with 6 neurons using linear and non-linear inputs ($swarm\_size = 100$, $\omega = 0.1$, $\alpha_1 = \alpha_2 = 2.0$)

2.3. Improving the baseline models to make PSO better in optimizing the weights

As shown in the literature, non-linear units make this problem a lot easier – choosing also X squared of *sin* transformation of X alongside it. We visualized our data in the playground and we think sinusoidal waves resemble the data the best, that's why we are going to use *sin* transformation. We also tried X^2 alongside X but we don't show the results here as it lead to slightly lower performance than with the sine one but much better than linear inputs anyway. Our experiments (Table 5) show highly increased performance for both **SGD** (the strongest baseline model was used when adding the non-linear inputs) and **PSO** compared to only linear inputs. Note that to get the final result in Table 5, we had to run our experiments multiple times to account for the weights initialization (starting point); the figures showed there are the ones from the most successful models with the best starting point. We also tried L1 regularization but the performance is slightly worse and does not really justify it for a simple model.

2.4. Designing a new fitness function

Having gathered a few very good models for this task, it is time to analyze how would a change in fitness function affect the performance. We want to explore the relation between the train and test error, therefore we want to also include that in our fitness computation for **PSO**.

Until now, we used the loss on train as the fitness function that we try to minimize. This is a poor choice of words though, as fitness is by definition a measure for how well fit an individual in a population is and we usually want those with the highest fitness. We can interpret this in our case as the fitness meaning actually: -$loss_{train}$ function for a candidate solution), so consequently, in this way, we try to maximize fitness.

2.4.1. Motivation and design

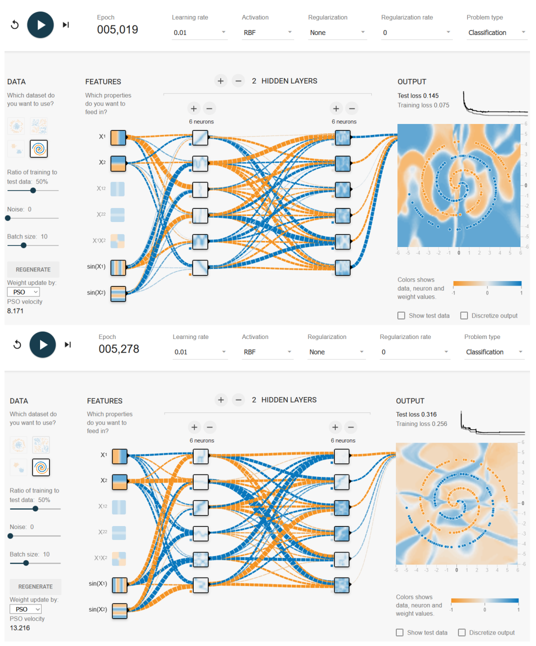
Many of our previous models suffered from overfitting and poor generalization. We are going to design a fitness function that improves on this particular matter using *MOO*. This directly targets the relation between test and train loss, we want the absolute value of the difference between those to be as small as possible, but let's not forget that the train loss should also be minimized as well. A first form of this new fitness function would look like this: $loss_{train} + |loss_{train} - loss_{test}|$. However, there is a fundamental problem with this, that being, we care about the minimization of these two objectives equally. Minimizing the loss train – the network actually learning patterns from the data should be the principal focus of the model, which is why we are going to assign a sub-unit weight to the second part: $loss_{train} + \lambda * |loss_{train} - loss_{test}|$. This is a form of scalarization in *MOO*.

What values should λ retain though? We tried with 0.1 and 0.01 with similar performances. There is no intuitive value and it depends completely on the task.

2.4.2. A theoretical improvement on the design

However, we designed **a variant of this fitness function** that may tackle the problem of choosing the **best value** for λ. This is beyond the scope of this study so we did not implement it but it is still worth to mention.

Usually, at the beginning of the training, we care about discovering the main obvious pattern that lead to decreasing the loss, that's why we shall choose a very small value at first (like 0.00001). Then, it **exponentially increases** directly proportional to the inverse square root of the number of steps. We do this, because we want the generalization performance to be as good as possible in the long run. This way, the trajectory of λ parameter would resemble an exponential function on a *loglog* scale.



Epoch 005,019 | Learning rate 0.01 | Activation RBF | Regularization None | Regularization rate 0 | Problem type Classification

DATA — Ratio of training to test data: 50% | Noise: 0 | Batch size: 10 | Weight update by: PSO | PSO velocity 8.171

2 HIDDEN LAYERS — 6 neurons, 6 neurons

OUTPUT — Test loss 0.145, Training loss 0.075

Epoch 005,278 | Learning rate 0.01 | Activation RBF | Regularization None | Regularization rate 0 | Problem type Classification

DATA — Ratio of training to test data: 50% | Noise: 0 | Batch size: 10 | Weight update by: PSO | PSO velocity 13.216

2 HIDDEN LAYERS — 6 neurons, 6 neurons

OUTPUT — Test loss 0.316, Training loss 0.256

DATA
Which dataset do you want to use?
FEATURES
Which properties do you want to feed in?
X1
X2
X1²
X2²
X1X2
sin(X1)
sin(X2)
REGENERATE
OUTPUT
Colors shows data, neuron and weight values.
-1
0
1
Show test data
Discretize output

Figure 1. PSO ($\omega = 0.1$; $\alpha_1 = \alpha_2 = 2.0$; $swarmsize = 100$; $\lambda = 0.001$) with RBF activation function and 2 hidden layers with 6 neurons. The fitness function for the first PSO is based only on the training loss and for the second one is the function we designed in 2.4.1.

2.4.3. New fitness function experiments

This new fitness function provides slightly higher accuracy performance on the best models and the generalization gap is much lower which is always a good thing (Figure 1). The model with the old fitness function achieved an accuracy of 96.4% and a gap between the train and test loss of 0.05 and the model with the fitness function we designed achieved an accuracy of 98.8% and a gap of 0.01. Although both of the models run for 5 epochs, these results may need to be taken with a grain of salt, as weights initialization might have played a role. We run it multiple times to assure that the results are consistent and most of the time, the accuracy values are comparable, however the gap between test and train loss always ends up lower in case of the new fitness function, which is to be expected.

2.4.4. Adding noise to the data

We decided to test the robustness of our best models adding a little noise to the data. The results are indeed consistent as adding some Guassian noise lead to slightly lower accuracy in our experiments.

3. Task 2 - Genetic Algorithms

After having found the best shape for our Neural Network through the experiments in the previous section, we are now going to use a **Genetic Algorithm** (GA) to optimise the shape of a NN and compare the results with the best shape we found.

For this task, our main concern is efficiency, since our fitness function uses the test loss after training on a certain number of epochs. This means that for a population of n individuals, each iteration would take n times the tim eto train for m epochs, which can be unpleasantly long if we don't account for that. That's why we decided to use **SGD** in our networks, which is much faster than **PSO** (and generally gives better results) with the *ReLU* activation function rather than *RBF*, which is slightly slower.

We decided to integrate our model with the code provided by the authors (https://github.com/tensorflow/playground) so that we would be able to relate our findings in this task with our conclusions from section 2. **We have implemented GA for this task from scratch without using any frameworks and integrated it intuitively in the playground.** It can be tested in the provided code.

3.1. Representation and restrictions

The individuals in our population are lists of possible shapes for a neural network. These individuals have some constraints: the first input layer can only have either 2 or 4 neurons ($[X_1, X_2]$ or $[X_1, X_2, Sin(X_1), Sin(X_2)]$; there is a maximum of 6 hidden layers and minimum of 1; each layer needs to have at least 1 neuron and no more than 8; the output layer can only have 1 neuron. All these restrictions are taken into account when doing the **crossover** and **mutation** operations described below. Both operations do work on a varying number of hidden layers and on varying number of hidden units. For example, an individual's genotype might be: $[2, 4, 1]$ or $[4, 2, 3, 4, 5, 1]$, meaning a NN with 1 hidden layer of 4 and linear inputs (for the first one) and a NN with 4 hidden layers and linear+nonlinear inputs, respectively.

3.2. Crossover

The **crossover** of two parent chromosomes is the main operator in **GAs** with a higher probability p_c and is carried out by swapping one segment of one chromosome with the corresponding segment on another chromosome at a random position (this is called *single-point crossover*) (Yang, 2014). For our GA, we are adapting this single-point crossover to our problem, that yields 2 offsprings at a time, introducing also some novelty that we think it would work best for this task.

In our GA, if both the individuals have more than 1 hidden layer, then we do single-point crossover with the middle position as the cut off point. However, this strategy does not work with one individual having exactly 1 hidden layer (length of 3), because the head and the tail would be just the input layer and output layer which have too few possible representations. That is why, if only one of the individuals has 1 hidden layer, then, the head would be the array composed by the first 2 entries in the individual's representation and the tail, last two entries. This results in offspring of mean size of the two parents. If both the individuals have 3 hidden layers, we have designed a special case. The first offspring will have average neurons between the first and the second individual and the second offspring will be just a replica of the one with the highest fitness. This way, we encourage both *exploration* (through the first offspring) and *exploitation* (through the second offspring).

3.3. Mutation

Mutation is usually done by flopping randomly selected bits (Yang, 2014). We will adapt this bit flip strategy for mutation in our **GA** because there is an obvious drawback to that - it lacks the diversity that comes with varying number of hidden units in a layer. In the upcoming section, we will also **prove** that for this specific problem, **mutation by itself is enough to obtain decent results**, that's why we are going to employ a mutation strategy a little bit more complex.

We use two types of mutation in our **GA**: one that modifies the dimension of the structure by adding or removing hidden layers (25% of the time) and one that modifies the neurons of a layer (75% of the time). For the other , we either add 1 neuron to a gene (layer) chosen randomly or subtract one, depending on the size of the layer and if the layer suffering the mutation is the input layer, then we simply swap between 2 and 4 neurons. This mutation strategy allows us to control the **number of hidden** units as well as the **number of hidden layers**, we want to favour the **exploration** for the hidden units because the crossover usually takes better care of the overall structure of the network.

During *next_generation* implementation, we do not apply mutations on the fittest individual because we are trying to stabilize the algorithm a little more and always keep the best solution at that point in order to display it. This individual can still be changed if more fit ones appear after the **crossover** and **mutation** operations on the rest of the population.

3.4. Selection

There are multiple choices of selection types we can do: Roulette selection, Tournament Selection, Elitism Selection, Rank Selection (Goldberg, 1989) (Miller et al., 1995). For our **GA**, we decided to chose a greedy approach, selecting the 2 most fit individuals from our population. We keep the size of the population constant across generations, hence, we will replace the two less fit individuals with the two new offspring. In some cases, due to crossover probability being different from 1, we wouldn't make any crossover, in that case, we just replace the 2 most unfit individuals with the two most fit ones, following the greedy approach that we want to employ for this method. We think that this will ensure **better convergence** as the generations pass and most important of all, it is a really **fast** approach and we do need that for this type of optimization.

3.5. Fitness

To evaluate the fitness of the individuals, naturally, we used the test loss of the corresponding NN trained with *SGD* after a certain small number of epochs, because it is the best indication of model's generalization power. The motivation behind this is the fact that the successful experiments for task 1 had the test loss decline really fast at the beginning and the improving only slightly and platteuing at around epoch 5. For the sake of the comparison with task 1, we are also going to consider 5 epochs to assert fitness, even though it is going to take longer time to train.

3.6. Population initialization

We initialize the population with shapes that only have 1 hidden layer and only have linear inputs for the first layer (2 neurons). This is because we want to show how fast the **GA** manages to improve over the different generations as well as the overall ascending trend in performance.

3.7. Tackling the problem of weights initialization

The weights initialization for Neural Network training is a research topic by itself and many studies were conducted on this area (Teo et al., 2001) (Glorot & Bengio, 2010). The starting point matters a lot and we actually observed that our best models for Task 1 did not work with the same hyper-parameters when all the weights are equal to 0 or 0.5. We did actually record divergence (for our experiments in task 1 we had a random weight initialization in the unit zero-centered interval). However, the weights initialization also had an effect on the PSO algorithm, although not that sever as in SGD's case, we were getting **unstable performance** for different weights initializations, but the performance was generally good (compared to SGD that might converge very prematurely or diverge). This problem has an effect on our GA algorithm as an individual can have a different fitness for the same genotype, if we reset the weights for each evaluation.

We tackle this problem by defining our goal to find the best individual that works across a various number of initial weights. The main idea is the following: define $t^k_{max} = max_i(t_{ik})$ as the best fitted individual with fitness value f^k_{max} at generation k. At each time before we calculate the fitness function training the NN for n epochs, we first reinitialize the NN with random weights. In this way, if at generation $k + 1$, the previous best individual does not preserve a certain consistency for the new initial weights of the network, meaning its fitness decreases drastically even though the genotype is the same as in the previous generation, his position as best individual will be replaced (assuming no other new individual raises at gen $k+1$ with a fitness even higher than t^k_{max}). Hence, for a certain individual to be really **the best**, it must yield the best fitness across all the random weights initialization (being consistent with any small change in the initial weights of the network). Therefore, we can be sure that the final individual would be the best fitted for our task and may even surpass what we found with grid search in task 1, at least consistency-wise.

3.8. Experiments

We first conducted some experiments to determine what the best parameters were. For the number of iterations we set 20 new generations to be made. We used our **GA** to find the best shape of NNs using an *RBF* activation function (which worked best for SGD) and $learning\_rate = 0.01$. The fitness of the individuals in the population was determined using the test loss of the corresponding NN after 1 epoch ($fitness = -test\_loss$). The results of varying p_c, p_m and the $population\_size$ are shown in table 6.

Varying Crossover Probability ($p_m = 0.15$, $population\_size = 16$)		
p_c	Fitness	Genotype
0.00	-0.081	[4, 3, 1, 2, 4, 1]
0.50	-0.135	[4, 1, 4, 1]
0.70	-0.102	[4, 2, 3, 3, 4, 1]
0.90	-0.075	[4, 4, 1, 1]
Varying Mutation Probability ($p_c = 0.90$, $population\_size = 16$)		
p_m	Fitness	Genotype
0.05	-0.099	[4, 2, 2, 2, 1, 1]
0.15	-0.075	[4, 4, 1, 1]
0.30	-0.088	[4, 2, 3, 3, 3, 2, 1, 1]
0.60	-0.139	[4, 4, 3, 1]
Varying Population Size ($p_c = 0.90$, $p_m = 0.15$)		
$population\_size$	Fitness	Genotype
8	-0.123	[4, 2, 4, 2, 1]
16	-0.075	[4, 4, 1, 1]
32	-0.092	[4, 2, 2, 2, 1, 1, 1]
64	-0.101	[4, 7, 1]

Table 6. Most fitted individuals - different parameters. No iterations: 20.

After the experiments we determined that the best parameters for our model are $p_c = 0.90$, $p_m = 0.15$ and $population\_size = 16$. What is interesting here is the fact that setting the probability crossover to 0 **(no crossover)** has very good results, very close

Activation function	Fitness	Genotype
ReLU	-0.150	[4, 1, 2, 1]
RBF	-0.075	[4, 4, 1, 1]

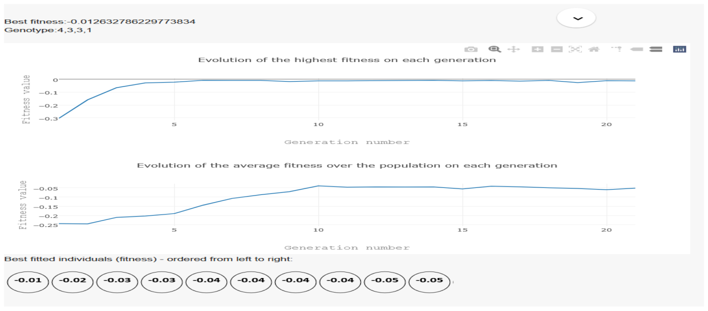
Table 7. Most fitted individuals - different activation functions.

Activation function	Number of Epochs	Fitness	Genotype
ReLU	2	-0.072	[4, 2, 4, 1]
RBF	1	-0.050	[4, 2, 1]
ReLU	5	-0.066	[4, 3, 1]
RBF	5	-0.013	[4, 3, 3, 1]

Table 8. Most fitted individuals - different activation functions using new fitness function.

to the usual standard value of close to 1, proving that the algorithm still works with only mutation. We then compared the results of training NNs with *ReLU* activation function and NNs with *RBF* activation function (table 7).

Finally, we tried to use our designed fitness function from task 1. To assert an individual's strength, $fitness = -(train\_loss + \lambda * |train\_loss - test\_loss|)$. We used lambda = 0.001 because it previously worked out best for us. The results are showed in table 8. For the best one, we find a NN (shape of [4, 3, 3, 1]) that is very similar to the best SGD NN we found in Task 1 [4, 6, 6, 1] with **grid search**. It is actually much better because the complexity of the NN is lower without compromising on the loss. (Figure 2).


Best fitness:-0.012632786229773834
Genotype:4,3,3,1

Evolution of the highest fitness on each generation

Evolution of the average fitness over the population on each generation

Best fitted individuals (fitness) - ordered from left to right:

-0.01 -0.02 -0.03 -0.03 -0.04 -0.04 -0.04 -0.04 -0.05 -0.05

Figure 2. GA ($p_c = 0.9$, $p_m = 0.15$, $population\_size = 16$) to find the best shape for a Neural Network trained for 5 epochs using the *RBF* activation function and $learning\_rate = 0.01$.

3.9. Conclusions

Using GA to optimize the structure of the NN over a large possible variants of model parameter initialization was really successful. Our algorithm ended up finding a better model than any we had before, beating grid search that found [4 6 6 1] which yielded 0.025 as test loss with a surprising [4 3 3 1] that reached 0.012 as test loss with is quite impressive (Figure 2). Moreover, we learnt that GA prefers quite simple models instead of complex ones with 5 or 6 hidden layers.

4. Task 3 - Genetic Programming

In this section we will try to solve the problem using GP (Banzhaf et al., 1998) and CGP (Miller, 1999). Just like for the previous task, we also integrated our model, which was built from scratch without using any frameworks, with the code provided by the authors of the google playground in order to achieve results that are consistent across the whole paper.

4.1. Overview and representation

It is regarded that neural networks are universal function approximation tools (Csáji et al., 2001). The main idea behind Neural Networks is the same as Cartesian GP. We don't know for sure what the model looks like so we are directly optimizing on the shape and the kind of the function that maps the inputs to the outputs and is defining our model. Using a graph representation is very flexible and can be used to model any kind of function. Neural nets are in a way, a particular case of **CGP**, as computational structures that can be represented as a string of integers. These integers, known as genes determine the functions of nodes in the graph, the connections between nodes, the connections to inputs and the locations in the graph where outputs are taken from (Miller, 1999).

4.2. Our approach

We are going to use 2 methods for this task: one that is closer to a classical Neural Network approach and one that resembles **CGP** better. This time, compared to **GA**, we are also optimizing the kind of the Neural Network for our classification problem. We got a set of many activation functions that can be applied to each node or a group of nodes. If we choose to apply the same activation to a group of nodes (layers), then this approach will be very close to how a NN operates, although, we are pretty much guessing what an appropriate activation would be for each layer.

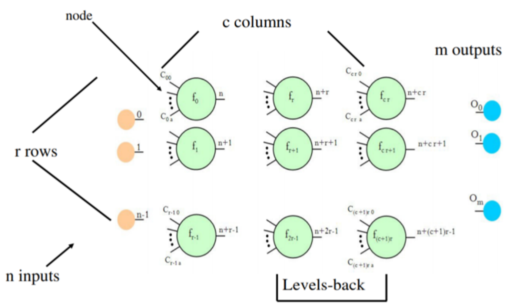


Figure 3. Cartesian Genetic Programming Schema (Mitchell, 1998)

.

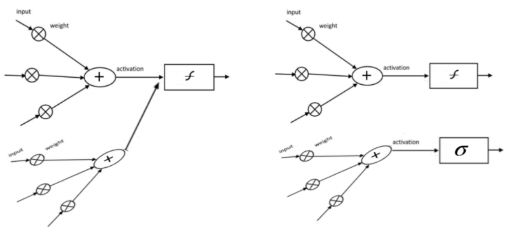


Figure 4. 1. First approach: like in a NN, we group neurons together and apply the transformation. 2. Second approach: we apply different activations to each neuron, even if for neurons in the same layer.

For our first solution, we will group nodes together and apply the same activation function for each one. Looking at the **CGP** general form (Figure 3), we can clearly observe that the general structure does resemble a modern-day neural network, however, there is a different activation function for each node. That's why for our second solution for this task, we will also model what transformation is going to be applied on each node's output.

It is important to note that all the products and summations that are done on the receiving end for a node are still present in our solution. What we optimize this time is the number of neurons (nodes), the number of neurons that we are grouping together (for solution 1) and what kind of activation is applied to each node/group of nodes. A visual explanation can be seen in Figure 4.

4.3. Initialization and Operations

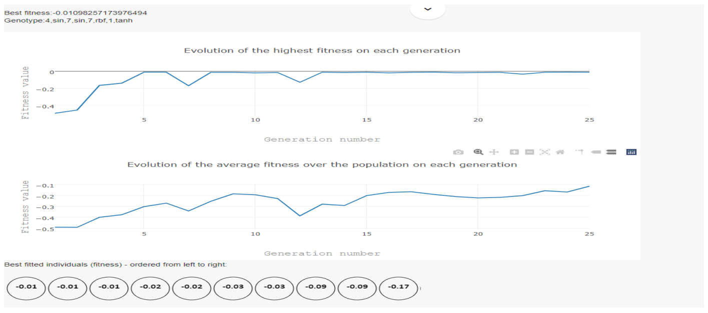
In addition to the hyper-parameters that we had with the classical GA in Task 2, now we got an extra one: the set of activation functions that can appear in an individual's representation. We want to make it hard for the algorithm to find an optimal solution in order to show the power of this method, that's why we chose almost all the available activation functions in the playground: *rbf, relu, linear, sin, tanh*. We first initialized all the activations for our starting population to be *linear*, as we will show that, indeed, the

GP and CGP choose non-linearities to get higher fitness values, which are, of course, necessary to solve this problem.

Just like in **GA**s, there are three main operations that lead to the evolution of the **CGP**'s genotypes: **mutation**, **crossover** and **selection**. The **implementation details** are **very similar** with those from the previous task, however, now an individual is represented as a graph that can be further narrowed down to a list of tuples (for easier implementation of crossover and mutation). The crossover now can merge sets of activation functions together (not just layers). The mutation now can also change one node's activation function, replacing it with another random one from the pool. The fitness function used is the same we defined in 2.4.1. Note that, for the individuals in our pool, we can look at one as being a higher order individual with some *"dead neurons"* – setting the output to that neuron to 0.

4.4. First approach

Compared to task 2, the search space is bigger in this task because now we have a pool of activation functions to choose from for each layer. A very low level representation of an individual could be represented by $[[2, 'rbf'], [3, 'rbf'], [1, 'sigmoid']]$, where we have an activation for each layer. In all our previous experiments, we had the same activation for each non-output layer so this is a massive improvement in terms of potential of the network and complexity.


Best fitness:-0.01098257173976494
Genotype:4,sin,7,sin,7,rbf,1,tanh

Best fitted individuals (fitness) - ordered from left to right:
-0.01 -0.01 -0.01 -0.02 -0.02 -0.03 -0.03 -0.09 -0.09 -0.17

Figure 5. Best first approach experiment. Test loss: 0.0109 after 5 epcohs.

There were 4 parameters to tune in our model: crossover probability (p_c), mutation probability (p_m), population size and the list of possible activation functions. After conduction some experiments to find the optimal parameters we have decided on $p_c = 0.9$, $p_m = 0.3$, $population\_size = 16$ and $activations = ['relu', 'rbf', 'linear', 'sin', 'tanh']$. After running the model for 25 iterations, we have achieved an average fitness of -0.12. The fittest individual had a fitness of -0.0109 in the end and its phenotype was a NN graph that can be summarized by: $[[4, 'sin'], [7, 'sin'], [7, 'rbf'], [1, 'tanh']]$ (Figure 5). We have also recorded the length for the best individuals' representation in each new generation: 2 had 3 hidden layers, 8 had 1 hidden layer and the rest (15) had 2 hidden layers. The max number of hidden layers possible is 6.

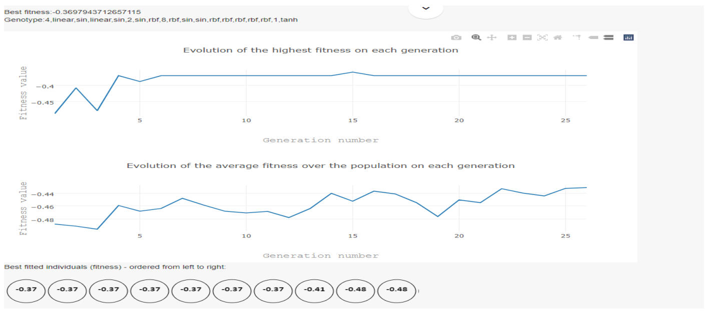
4.4.1. Remarks

It is really interesting that this algorithm beat our previous best (it obtained a higher fitted individual than GA). Needless to say, this also beat our grid search from task 1. The algorithm immediately introduces and keeps **non-linearities** (even though it had only linearities at first), this shows that our model is learning correctly to solve the problem. We observed that the algorithm also preferred the **shorter architecture** with more nodes on a group. Moreover, a population of 16 is actually not needed, we've tested with population **length of 5** as well and it works similarly, plus, figure 5 suggests that as we can see, the fitness of top 5 individuals is very close to one another. Finally, **crossover is not needed**, the model with 0 as crossover probability actually found a solution with test loss of 0.009, on generation 24, which is higher than everything we got before, it finished with a test loss of 0.01 for the fittest individual, which is still really good. Note that we needed to **increase the mutation probability** compared to the task 2 experiments with GA. We run with 0.15 as mutation probability, but often the algorithm failed to discover the high end solutions, there wasn't enough diversity, that's why we decided to double that value.

4.5. Second approach

This time, the search space increases even more, as we allow different activation for each neuron, this might result in highly polarized, unstable performance for our genetic approach. Note that now, we have a computational structure that does not match a neural network as we are using **CGP** to build a graph that relates to a function to convert the input features to a probability for our binary classification task. It should come as no surprise that this algorithm might converge to a local minima if the number of iterations of crossover/mutation probabilities are not set well enough. We can interpret an individual this time as list of tuples, although the list can get very big: [[2,["rbf","linear"]],[3,["relu","sin","rbf"]],[1,"sigmoid"]].

We conducted several experiments to find the optimal values for the hyper-parameters and landed on $p_c = 0.9$, $p_m = 0.3$, $population\_size = 16$ and $activations = [relu, linear, sin]$. After running the model for 25 iterations, we have achieved a best fitness of -0.36 (Figure 6). This constitutes significantly worse results than the ones from our first approach, however this was expected since there is a significant increase in the search space and we need more iterations or a higher population size to match the previous results.


Best fitness:-0.3697943712657115
Genotype:4,linear,sin,linear,sin,2,sin,rbf,8,rbf,sin,sin,rbf,rbf,rbf,rbf,rbf,1,tanh

Best fitted individuals (fitness) - ordered from left to right:
-0.37 -0.37 -0.37 -0.37 -0.37 -0.37 -0.37 -0.41 -0.48 -0.48

Figure 6. Best second approach experiment. Test loss: 0.36 after 5 epcohs.

4.5.1. Remarks

As expected, this time, we saw a real increase in the number of hidden layers, for the best model, for the best fitted individual in each new generation, we got: 3 individuals with 6 hidden layers, 3 with 5 hidden layers , 13 with 4 hidden layers and the rest with less or equal than 3 hidden layers. It seems long-thin architectures are better than short-thick ones. Similar performance can be achieved with a population of 5 as well, we got a max fitness of -0.2 at one point but it was much more unstable. To preserve consistency we also run with crossover probability of 0 and we got the best fitness of -0.382, similar result to our best CGP model.

4.6. Further work

Other tasks that would be interesting in this context would be extending the scale of the experiments. The problem was quite simple, being easily solvable with really small NN architectures. We were also really constrained by the playground implementation, JS is not the fastest language for these kind of experiments, but it is rather good for visualization. It would be interesting to see if we can get the test to 0 after a higher number of epochs as we know this is possible using frameworks in other languages like python. However, GA and GP still provide a good optimization to finding the optimal architecture, compared to the brute force ones.

5. Conclusion

Through our tasks, we were able to understand how to use nature inspired techniques on neural networks for a classification task. We started by comparing Particle Swarm Optimisation and Stochastic Gradient Descent in the task of training a neural network and concluded that, with careful initialisation and parameter choice, they can perform at the same level. Our last two tasks allowed us to conclude that both Genetic Algorithms and Genetic Programming are two useful techniques for hyper-parameter tuning and are both capable of finding better results than a grid-search. Moreover, GP managed to provide slightly better results than the GA.

References

Alvarez-Sanchez, JR. Injecting knowledge into the solution of the two-spiral problem. *Neural Computing & Applications*, 8(3): 265–272, 1999.

Banzhaf, Wolfgang, Nordin, Peter, Keller, Robert E, and Francone, Frank D. *Genetic programming*. Springer, 1998.

Binkley, Kevin J and Hagiwara, Masafumi. Balancing exploitation and exploration in particle swarm optimization: velocity-based reinitialization. *Information and Media Technologies*, 3(1):103–111, 2008.

Csáji, Balázs Csanád et al. Approximation with artificial neural networks. *Faculty of Sciences, Etvs Lornd University, Hungary*, 24(48):7, 2001.

Eberhart, Russell and Kennedy, James. Particle swarm optimization. In *Proceedings of the IEEE international conference on neural networks*, volume 4, pp. 1942–1948. Citeseer, 1995.

Glorot, Xavier and Bengio, Yoshua. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256, 2010.

Goldberg, David E. Genetic algorithms in search. *Optimization, and MachineLearning*, 1989.

Holland, John H. Genetic algorithms. *Scientific american*, 267(1): 66–73, 1992.

Karlik, Bekir and Olgac, A Vehbi. Performance analysis of various activation functions in generalized mlp architectures of neural networks. *International Journal of Artificial Intelligence and Expert Systems*, 1(4):111–122, 2011.

Kennedy, Jim and Eberhart, Russ. Tutorial on particle swarm optimization. In *2002 World Congress on Computational Intelligence WCCI*, 2002.

Lang, Kevin J and Witbrock, Michael J. Learning to tell two spirals apart. In *Proceedings of the 1988 connectionist models summer school*, number 1989, pp. 52–59. San Mateo, 1988.

Miller, Brad L, Goldberg, David E, et al. Genetic algorithms, tournament selection, and the effects of noise. *Complex systems*, 9(3):193–212, 1995.

Miller, Julian. *Cartesian Genetic Programming*. 1999.

Mitchell, Melanie. *An introduction to genetic algorithms*. MIT press, 1998.

Neal, Radford M. Connectionist learning of belief networks. *Artificial intelligence*, 56(1):71–113, 1992.

Nwankpa, Chigozie, Ijomah, Winifred, Gachagan, Anthony, and Marshall, Stephen. Activation functions: Comparison of trends in practice and research for deep learning. *arXiv preprint arXiv:1811.03378*, 2018.

Teo, Kok Keong, Wang, Lipo, and Lin, Zhiping. Wavelet packet multi-layer perceptron for chaotic time series prediction: effects of weight initialization. In *International Conference on Computational Science*, pp. 310–317. Springer, 2001.

Yang, Xin-She. *Nature-Inspired Optimization Algorithms*. Elsevier, 2014.