Natural Computing Assignment 2

s1610357, s1721204

November 19, 2020

Task 1: Particle Swarm Optimisation (PSO)

Our fitness function for this problem was the Binary Cross-Entropy (BCE) loss function. We chose this as it was a simple cost function to implement that is standard in classification problems. We did not feel it was worth increasing the complexity of this as it drastically affects the computational cost. We evaluated results by their loss and accuracy along with how much the fit looked like a spiral. Generally, the loss was enough to determine performance but in closer results the shape of the fit helped decide.

For deciding our search space, we ran some baseline tests on the tensorflow playground [1] interactively. We found that 1 hidden layer and 8 neurons gave the best result. This left a search space of $\mathbb{R}^{49}$ in which we allowed the weights to take any value and relied on the regularization to maintain them. Only the x, y, x^2, y^2 features were used as we saw no significant improvement when using $sin(x)$, $sin(y)$ and decided against using these to save on computational cost. We found that the test loss began to plateau at around 1000 epochs; ReLU activation was used with L2 regularization of 0.003. We kept the noise at 0 for a baseline, although at the end of this report we also investigate the effects of noise. We also implemented stochastic gradient descent (SGD) as a second baseline. We ran some initial tests and found our PSO could run 30 particles for 1000 epochs in about 40 minutes and decided this was acceptable as we could run batches of it overnight. We found that the particles moved too far relative to the scale of the weights, so we divide the position by 1000 before using it as a weight vector.

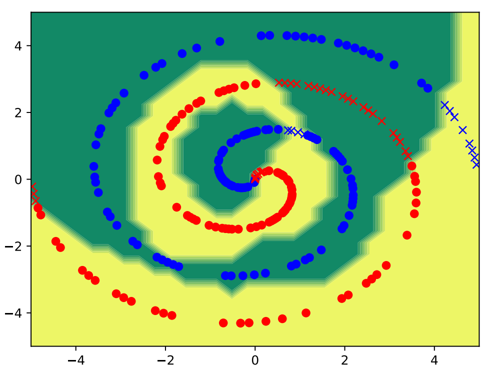
Our parameters were adjusted from a_1, a_2, w to $a, a_{1,\%}, \epsilon$. Where:

$$a = a_{1,\%}a_1 + (1 - a_{1,\%})a_2 \tag{1}$$
$$w = \epsilon + a + 1 - \sqrt{4a} \tag{2}$$

This meant we were able to define our parameters in reference to the region of complexity, as ϵ controls how far we are from the boundary of complexity.

Our playground testing gave a target of 0.1 test loss in 1000 epochs which is quite a good fit on a notoriously hard problem. SGD, for a batch size of 30, 1000 epochs and averaging over 100 runs gave an average loss of 0.211 with a deviation of 0.027. Individual results could vary a lot, but the average of all these runs gave an impressive baseline.

To find our optimal parameters. we carried out a greedy grid search. By setting our value of ϵ such that $w \approx 0.7$, we varied a from 2 to 4.04 (the value we used in CW1). Note we kept $a_{1,\%} = 0.5$ throughout. We found the best results for $a = 3.4$, at which we then varied ϵ. Our best result was found at $a = 3.4$, $\epsilon = -0.21$ ($w \approx 0.5$) which gave a test loss of 0.507, the fit can be seen in Figure 1a. We decided to try allowing this to run further to see how good of a fit we can achieve. In 2000 epochs we got a test loss of 0.396 and the result is shown in the appendix.
When deciding the search space when using only linear features we knew it would have to be more complex in order to successfully replicate the spiral pattern. However, this would significantly increase computational cost, so we began with only adding one extra hidden layer. We found that altering the structure in playground to 2 layer of 6 neurons each, and keeping


(a) PSO results for $a = 3.4, \epsilon = -0.21$, population=30, with 1 hidden layer of 8 neurons using nonlinear features.

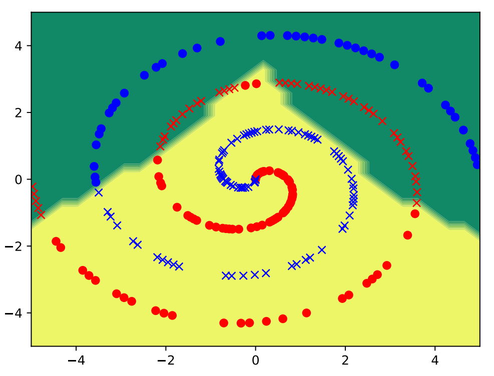

(b) PSO results for $a = 3.9, \epsilon = -0.25$, population=30, with 2 hidden layers of 6 neurons each using only linear features.

Figure 1: 1000 epochs. Dots imply correctly classified points and xs are incorrectly classified points.

all other parameters the same, still gave excellent results, reaching a test loss of less than 0.15 in 1000 epochs. We then chose this as our PSO search space as it was the simplest model we found with only linear features that gave a good result. Running SGD as before gave a test loss average of 0.267 with standard deviation of 0.074.

We repeated the same search over a, with all other parameters the same (for consistency) and found $a = 3.9$ gave the best fit. We then held a constant and varied ϵ. Our best result was for $a = 3.9, \epsilon = -0.25$ ($w \approx 0.7$), giving a test loss of 0.709, the resulting fit is shown in Figure 1b. Again, we decided to see how good the fit could run in 2000 epochs. This achieved a test loss of 0.661 and the result is shown in the appendix. PSO performed significantly worse here compared to the more simple network and also in comparison to the playground and SGD baselines. This demonstrates the difficulty PSO has in finding solutions to solutions with large search spaces.

The hyper-parameters of PSO control the exploration (or exploitation) of the solutions in the search space. Given the large size of the search space here we require a moderate level of exploitation to achieve a solution in a reasonable time. However, the weights in neural networks are very sensitive and are usually at small values (therefore small changes create large effects) and so we must have restrained parameters in order to actually be able to find a good solution (once we are near one). This is reflected in our results for the simpler model as for extreme values of a and w we got poor results, while our best result is for moderate parameters. In the second more complex model the more exploratory parameters got a better result due to the significantly larger search space requiring a much wider exploration to find parameters that give any useful solution. It is probable that a constriction factor would help a lot here and given more time this is the next step we would try to implement for this specific problem.

Our PSO results were outperformed by the SGD baseline in both sections and took significantly longer to run. While individual SGD steps may have widely varying performance, in a longer run they achieve a better result and faster. The playground results were also much better than our PSO results. However, the playground results required understanding of the effects of features/neurons/layers/learning rates etc while all we required in PSO was tuning of a few searching parameters. This greatly simplified the problem and allowed us to perform a somewhat ‘uninformed’ search in a defined space, which is hugely beneficial in many problems, particularly when the problem is not yet fully understood. Due to the performance of SGD, we decided to use it in our implementation of GA and GP.

Task 2: Genetic Algorithm (GA)

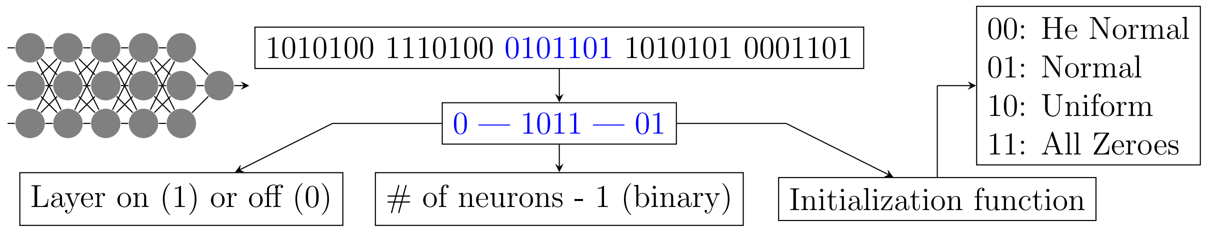
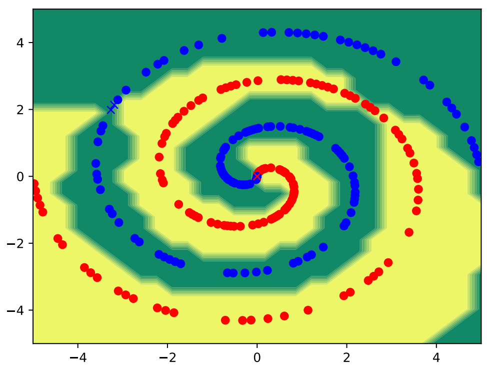
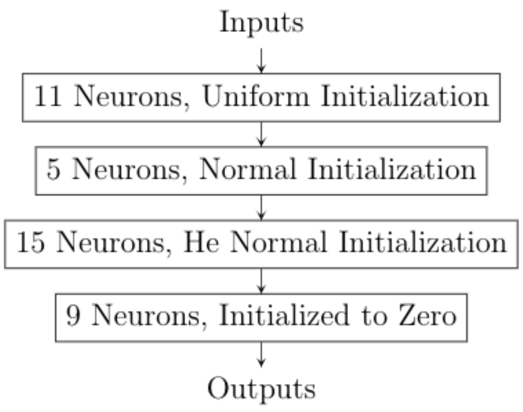


Figure 2: The genotype consists of 7 sets of 5 bits, each controlling an individual layer as described in this diagram. We use mutation and crossover as operators as they are standard for GAs.

We chose to have a bit represent whether a layer is active or not because it makes it easy for mutations to remove/add a layer if necessary. Removed layers should accumulate mutations (as they do not affect fitness, so no selective pressure to keep them in place) - this should encourage exploration, assuming the layer gets turned on again eventually. We have elitism; the best network survives each generation unchanged. We chose this because GA takes relatively long to run, so we cannot run it for many generations on our computers. Every step backwards in fitness is a major loss for us under these restrictions, which elitism prevents.



(a) Decision Boundary of Best Model


Inputs
11 Neurons, Uniform Initialization
5 Neurons, Normal Initialization
15 Neurons, He Normal Initialization
9 Neurons, Initialized to Zero
Outputs

(b) Structure of Best Model

Figure 3: GA best result (Population: 30, Mutation: 0.1, Crossover: 0.8). Loss: 0.151, Accuracy: 0.988.

We restrict crossover to only happen at the start of one of the 5-bit layer specifications, as this seemed more sensible for the genotype structure and appeared to give better results. We implemented a coarse grid search again to find the best settings before carrying our a longer run at those settings. Once we found a good setting for the parameters, we used a population of 30 for 100 time steps. The best result gave a loss of 0.151 (Figure 3). However, most settings gave roughly the same loss, except when both crossover and mutation were high, which is expected as there would be too much exploration to find a close solution. That being said, the worst result still had a loss of about 0.4. The similar results for most settings is likely due to roughly similar levels of exploration/exploitation. Overall, the performance of the GA was excellent and only ran slightly slower than the PSO code. Our mutation and crossover rates are higher than the standard values (0.01 and 0.5) due to our small population and short run time. This ensures enough diversity for adequate exploration under these constraints.

We control complexity by measuring fitness on the validation set rather than training; complicated networks will tend to overfit to the training set and thus do worse on the validation set, meaning their fitness is lower. Our best network does not use all 5 possible layers, which is likely a result of this. The layer initialization function does not seem to matter much, and it seems to alternate between thick and thin layers; it would be interesting investigate whether there is a reason behind this in future experiments.

Task 3: Genetic Programming (GP)

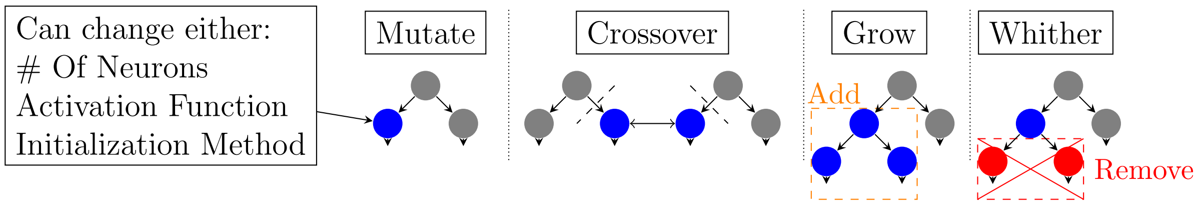
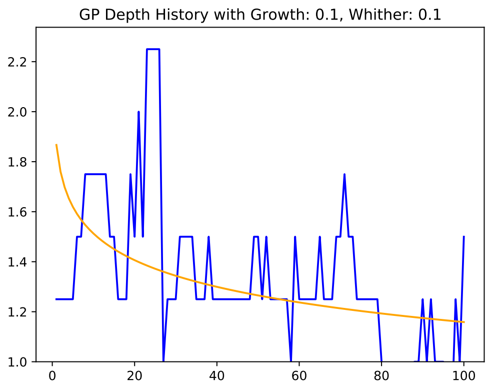
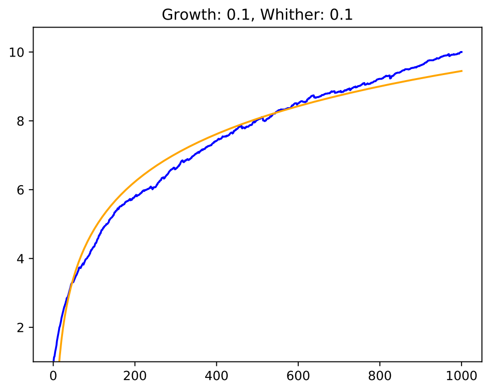


Figure 4: The operators of GP. Each node in the tree is a 'layer' of the neural net, and can consist of multiple neurons, its own activation function, and its own initialization method. Nodes can have at most two children. It can choose any activation function available in Playground, as well as some others. Unlike GA, mutations are on a per-organism basis rather than per-bit (i.e. mutation=0.5 means there is a 50% chance for a mutation to trigger, and then we select a node to mutate).

We chose these operators because they are fairly standard and make it easy to control the diversity of the population; the results suggested adding further operators was unnecessary. The whither and growth operators in particular interact with each other in an interesting manner. To study this, we measured what the average tree depths for given whither/growth parameters would be if there were no fitness function. Depending on the parameters, the fitness-less depths would either asymptote to a small value, or increase without bound. By comparing this to tree depths measured during an actual GP run, we could get a better sense of whether the model was selecting for shallow or deeper trees.



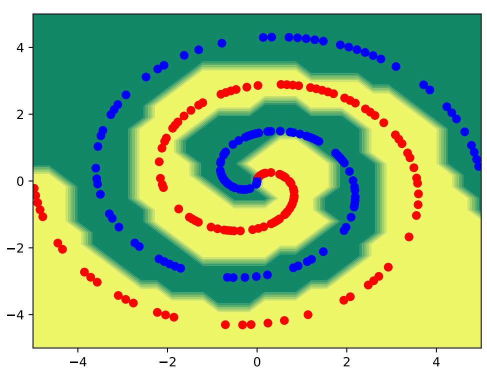
(a) GP Average Tree Depth



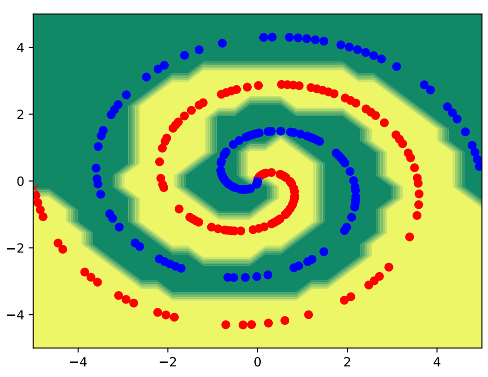
(b) Average Tree Depth Without Fitness

Figure 5: Orange line is a logarithmic fit. As we can see, the depth is much shallower than would be expected if fitness had no role in limiting the depth. This is likely because we measure fitness by the validation error, which acts a bit like regularization over tree depth in that it prefers less complex models that won't overfit

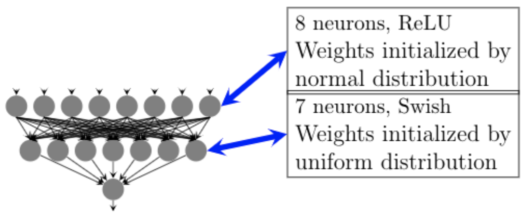
We used a coarse grid search before doing longer tests on the best performing settings. (which had a batch size of 20, 100 time steps, 200 train epochs and 1000 test epochs). The performance varied more as these values were altered than with GA. The best result with random initialization had a loss of 0.043, shown in Figure 6a. This is by far the best result we have achieved and even beat the interactive playground (although was given much more time to run) with a nearly perfect fit. Like GA, our mutation rate is higher for similar reasons and also because our mutation method in GP is per organism rather than per bit, requiring it to be higher to ensure enough exploration in the search. We found that no crossover gave the best results, and that large population sizes were unnecessary, similar to what is expected for Cartesian Genetic Programs (CGP). Our implementation preferred short/thick networks, contrary to expectation. This is probably due to problem complexity (or lack thereof) in our setup; the PSO results show that we do not necessarily need much more than one layer for this problem. This lack of complexity comes from the lack of noise and using non-linear features to simply the fitting. If we were to change this it is possible the network shape may change to a longer and thinner style. However, this would require testing to confirm.



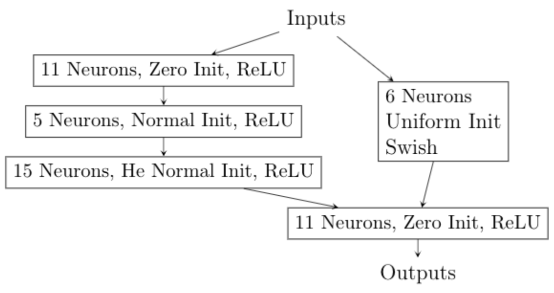
(a) Random initialization.
Loss: 0.043, Accuracy: 0.996.



(b) GA best as initialization.
Loss: 0.032, Accuracy: 0.992.


8 neurons, ReLU
Weights initialized by normal distribution
7 neurons, Swish
Weights initialized by uniform distribution

(c) Structure of random init network


Inputs
11 Neurons, Zero Init, ReLU
5 Neurons, Normal Init, ReLU
15 Neurons, He Normal Init, ReLU
6 Neurons Uniform Init Swish
11 Neurons, Zero Init, ReLU
Outputs

(d) Structure of GA init network

Figure 6: GP best results (Population: 4, Mutation: 0.5, Crossover: 0, Whither: 0.1, Growth: 0.1)

When we initialized our GP forest using the structure of the best GA network, we achieved an even lower loss of 0.032. This is likely because it is easier for GA to find a large good network than GP (the 'whither' operator in GP is quite effective at shortening the network compared to GA which requires specific mutations to lower size). If we look at the history of GP performances per timestep, when initialized to GA it finds a better network within the first 10 timesteps (see Figure 11b in the appendix), but then makes no further improvements (unlike random initialization which is improving continuously); this suggests that the we are at the peak performance in this local region, a region that is hard to access from random GP initialization.

Interestingly, with the GA best as the initialization in GP we achieved a marginally lower accuracy while also lowering the loss. The accuracy dropped from 0.996 to 0.992 with the change of initialization. This is very minimal and so possibly insignificant as only one run was carried out. However, this decrease in both loss and accuracy can be reasoned by the shape and smoothness of the spirals. With higher accuracy more points are classified correctly but with lower loss the boundaries are smoother and better fit to a perfect spiral in zero noise data. This is reflected in Figures 6a and 6b. GA/GP hybrid algorithms seem to be a promising place for further experiments.

Future Experiments

To improve PSO, we could have implemented variable parameters (such as via a constriction factor), which reduce after a number of epochs or when a certain loss was achieved to enable a faster initial search. Perhaps we might also try scaling inertia depending on how much performance improved in the last timestep, similar to how 'momentum' works in gradient descent methods. As mentioned previously, it would be interesting to study whether there is a reason why GA seems to prefer alternating thick/thin layers. We also believe that an appropriate next step would be to study the robustness of the models when faced with noisy data. We have made a start to studying this as we had extra time and the preliminary results can be seen in the next section with a short discussion. Finally, we would also like to investigate the performance of GA/GP hybrid algorithms in general, as our best performance was achieved by initializing the GP forest to the output of GA.

Add-on: The Effect of Noise

We wanted to also study the robustness of the methods when faced with a noisy dataset. We added Gaussian noise to each point with variance of 0.5 to produce the data. We then ran the best parameters from each method stated previously. The results can be seen in Figure 7 in the appendix. GP still out performs the rest, with GA in second. However, PSO now performs better than SGD. It has a marginally better loss but notably the fit has a much more spiral shape, compared the to a more concentric ring fit in SGD. This is expected as PSO learns the global structure of the data (thanks to having multiple particles) more than SGD, which follows a random gradient at each step, and so would be expected to deal with some noise better. The power of these meta heuristic methods is demonstrated here, each learning a difficult spiral system with quite moderate noise.

For further study with noise, varying the level of noise to find a limit on how much the models can deal with would be interesting. Additionally, studying the performance of the GA best with noise as the initialization of GP, which managed to lower the loss marginally previously, could produce an even better fit here. Finally, as PSO is clearly better at learning from and dealing with noise than SGD, with more time we could have potentially implemented the use of PSO instead of SGD in our GA and GP programmes to perhaps reduce the loss in noisy data fit even further.

Reproducability

If the reader wishes to use our code, it was submitted as a zip file along with this report as part of assignment submission. The script `main.py` takes a path to a text file as an argument, with the text file including a list of algorithms to run and their parameters. The format for this text file is intuitive and is explained in the `README.md`. In the appendix we have put an example text file to reproduce our best results.

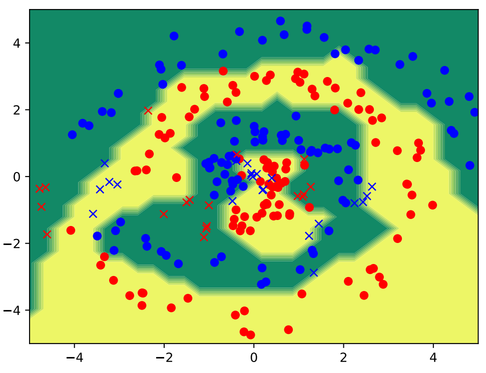
References

1. Martın Abadi *et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems* Software available from tensorflow.org. 2015. `https://www.tensorflow.org/`.

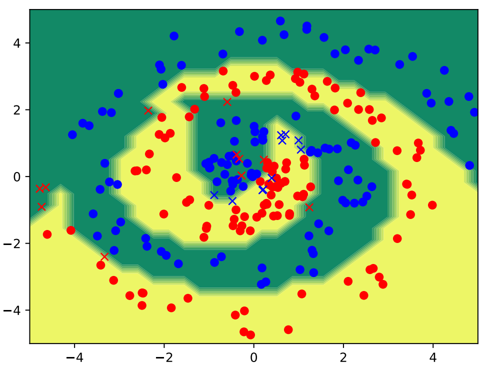
2. Ritchie, M. D., Motsinger, A. A., William S Bush, C. S. C. & Moore, J. H. Genetic Programming Neural Networks: A Powerful Bioinformatics Tool for Human Genetics. *NCBI* (Jan 2007).

Appendix

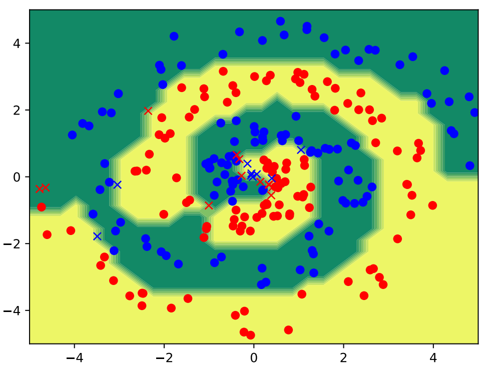
Noise Results



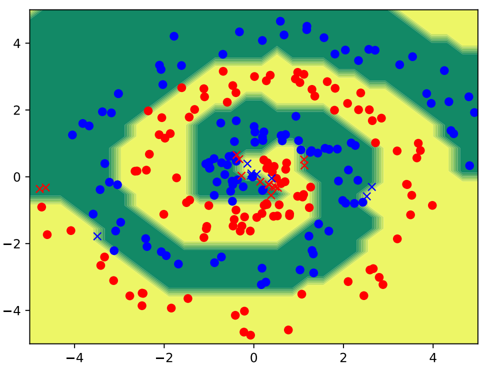
(a) SGD: loss=0.445 ± 0.037



(b) PSO: loss=0.427, accuracy=0.908



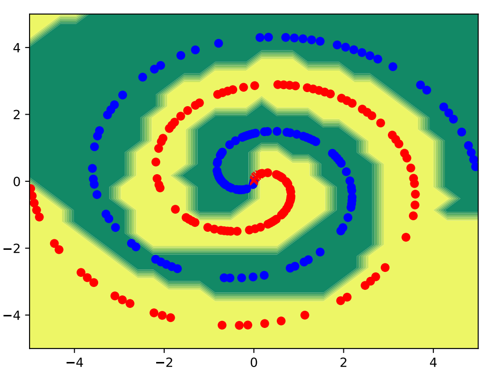
(c) GA: loss=0.346, accuracy=0.908



(d) GP: loss=0.217, accuracy=0.904

Figure 7: Noise results with the best parameters discussed for each setting.

SGD Figures of Best Results


(a) 1000 epochs

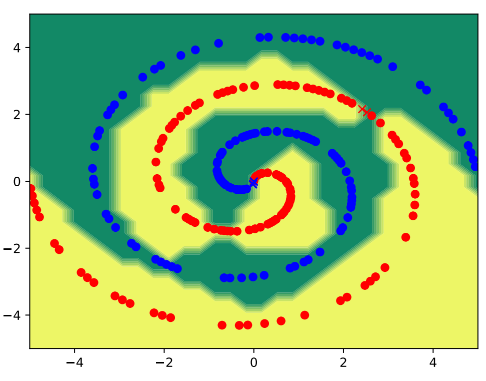
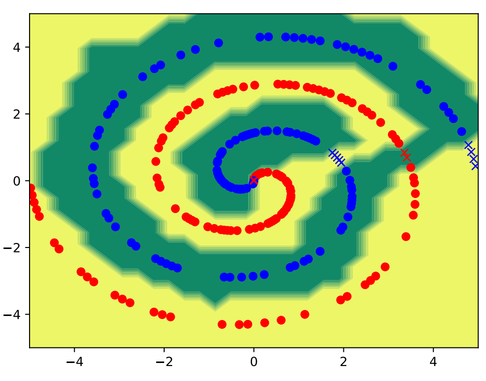

(b) 2000 epochs

Figure 8: SGD results with population=30 averaged over 100 runs, with 1 hidden layer of 8 neurons using x, y, x^2, y^2 features. For numeric results see the table on the next page.


(a) 1000 epochs

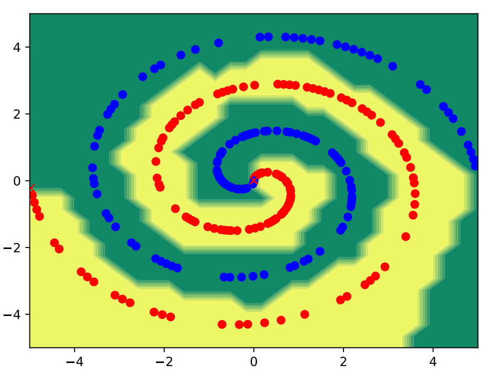
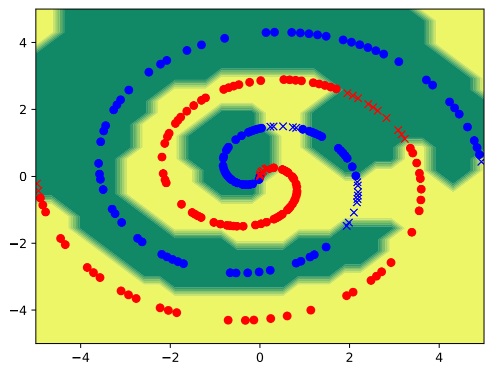
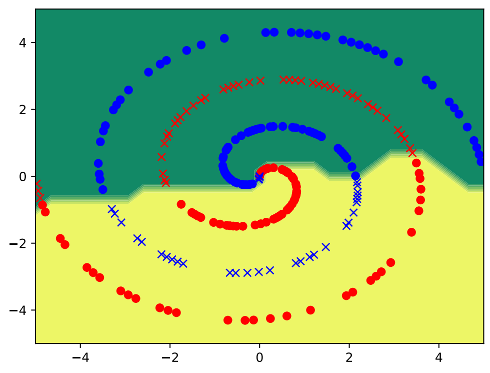

(b) 2000 epochs

Figure 9: SGD results with population=30 averaged over 100 runs, with 2 hidden layer of 6 neurons each using linear features. For numeric results see the table on the next page.

PSO Performances at 2000 Epochs



(a) PSO results for non-linear.



(b) PSO results for linear.

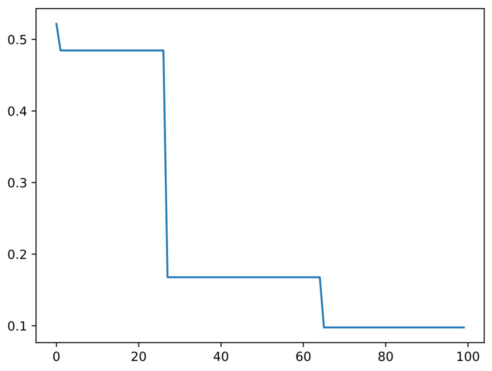
Figure 10: PSO for 2000 epochs. For numeric results see the table below.

Tabulated Best Results for Zero Noise

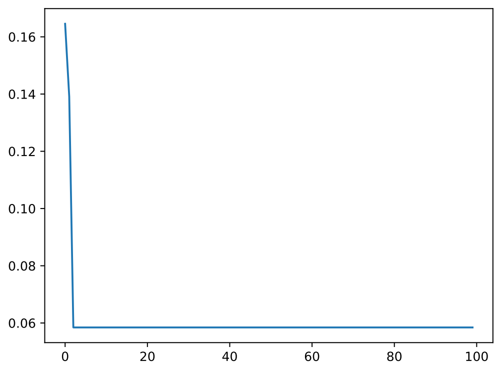
			Loss	Accuracy
PSO	Non-linear	1000 epochs	0.507	0.816
		2000 epochs	0.396	0.852
	Linear	1000 epochs	0.709	0.488
		2000 epochs	0.661	0.692
SGD	Non-linear	1000 epochs	0.211 ± 0.027	0.975 ± 0.011
		2000 epochs	0.258 ± 0.020	0.988 ± 0.011
	Linear	1000 epochs	0.267 ± 0.074	0.969 ± 0.054
		2000 epochs	0.187 ± 0.004	$1.000 - 0.001$
GA			0.151	0.988
GP		Random initialization	0.043	0.996
		GA best as initialization	0.032	0.992

Table 1: Settings for best results:
PSO Non-Linear: $a = 3.4, \epsilon = -0.21$, pop= 30
PSO Linear: $a = 3.9, \epsilon = -0.25$, pop= 30
SGD Non-Linear and Linear: batch size=100, pop= 30
GA: pop=30, mutation rate=0.1, crossover rate=0.8, train epochs=200, test epochs=1000, batch=20
GP: pop=4, mutation rate=0.5, crossover rate=0.0, flip chance=0.5, whither rate=0.1, growth rate=0.1 train epochs=200, test epochs=1000, batch=20

GP Performances Over Time



(a) Random Init Loss Over Time



(b) GA Init Loss Over Time

Figure 11: We can see that GA init very rapidly reaches the best value it ever will. Its losses are incredibly small already (about 17x as good as PSO for 1000 epochs!) so it makes sense that it would have trouble finding further improvements. The random init also improves in 'big bursts', but these happen multiple times over the course of the run. This is for a similar reason; GP is already doing just about perfectly. If the task were a more complex one, we would expect these graphs to be more gradual, as they wouldn't start out near peak performance. Chance plays perhaps just as big a role as the algorithm's parameters when we are so close to perfect performance.

GP Inspiration

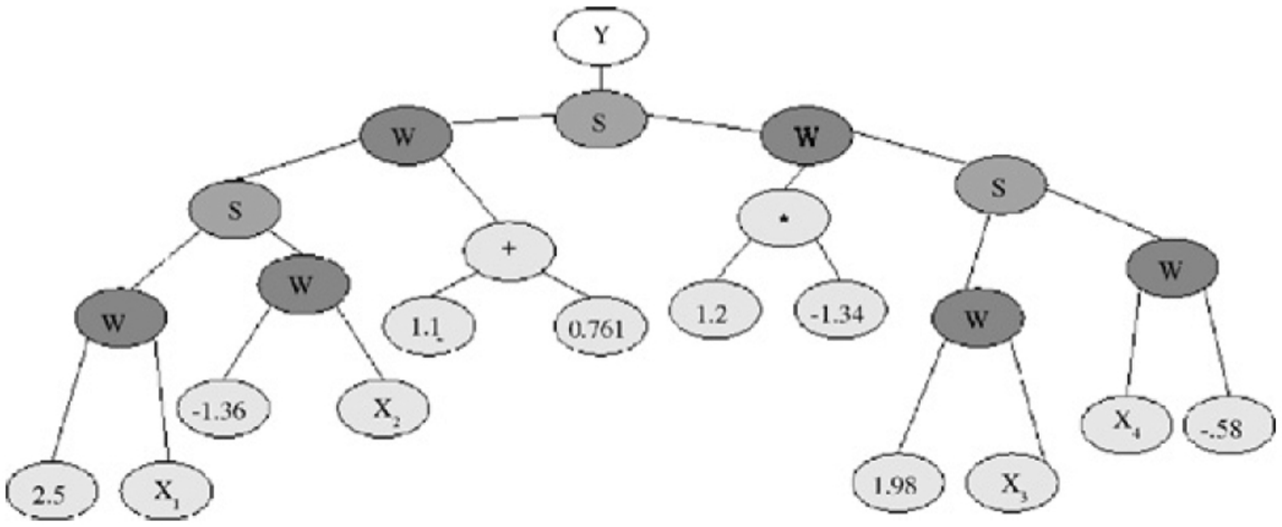


Figure 12: Our GP algorithm's architecture was inspired by this image, from a paper [2] on GP used to train neural networks for bioinformatics problems. (Their algorithm also trains the weights and uses nodes to represent a single neuron, whereas ours does not train the weights and uses nodes to represent a layer of neurons.)

How to run

Run `python3 main.py Sample_inputs.txt` to run a batch that will reproduce all of our numerical results (except GP initialized to GP), and most of our graphs. To get the fitnessless tree depth graphs, run `python3 tree_utility.py`. To see how GP does when initialized to the best GA network, run `python3 gp_initialize_to_ga.py`.

Below we have also added an example text file (this is just the aforementioned `Sample_inputs.txt` that comes as part of our repository). Information on how to edit this file if you want to run your own experiments is given in the `README.md` of our repository.

```
COMMENT; Change '0' to whatever you want to increase the amount of
    noise. We used 0.5 for our noise tests.
COMMENT; You might want to decrease time_steps for each of these as
    it will take hours to run! But this should replicate our results
    , up to machine-dependent differences in RNG.
DATA_PARAMETER; NOISE:F0
SGD; population_size:I30, time_steps:I1000, averaging:I100
PSO; total_a:F3.9, a1_percent:F0.5, epsilon:F-0.21, population_size:
    I30, time_steps:I1000, a3:F0
GA; population_size:I30, time_steps:I100, mutation_rate:F0.1,
    crossover_rate:F0.8, train_epochs:I200, test_epochs:I1000, batch:
    I20
GP; population_size:I4, time_steps:I100, mutation_rate:F0.5,
    crossover_rate:F0.0, flip_chance:F0.5, whither_rate:F0.1,
    growth_rate:F0.1, train_epochs:I200, test_epochs:I1000, batch:I20
```