Biologically Inspired Computing
-
Training an Artificial Neural Network using Particle Swarm Optimisation
-
https://www2.macs.hw.ac.uk/~sf52/Bio-Comp-docs/html/index.html

Sam Fay-Hunt — sf52@hw.ac.uk
Kamil Szymczak — ks83@hw.ac.uk

November 22, 2020

Contents

1 Introduction

This report details our reationale when developing our Artificial Neural Network (ANN) and Particle Swarm Optimisation (PSO) implementations, A description of the methodology we employed, and the results from the experiments we performed to gain insight into the factors that can effect the quality of PSO.

Our solution is written in Python, heavily utilising numpy, pandas and the python standard libraries, additionally we made use of matplotlib for plotting graphs and SKlearn to split the data into training and testing sets. We used an OOP approach to keep the project organised so we could maintain the fairly large codebase, this also neatly decoupled the ANN and the PSO modules. We used jupyter notebooks to demonstrate how to use our codebase, and to present our findings. We used sphinx to generate documentation of the sourcecode, links to our documentation are included throughout this report to help provide context.

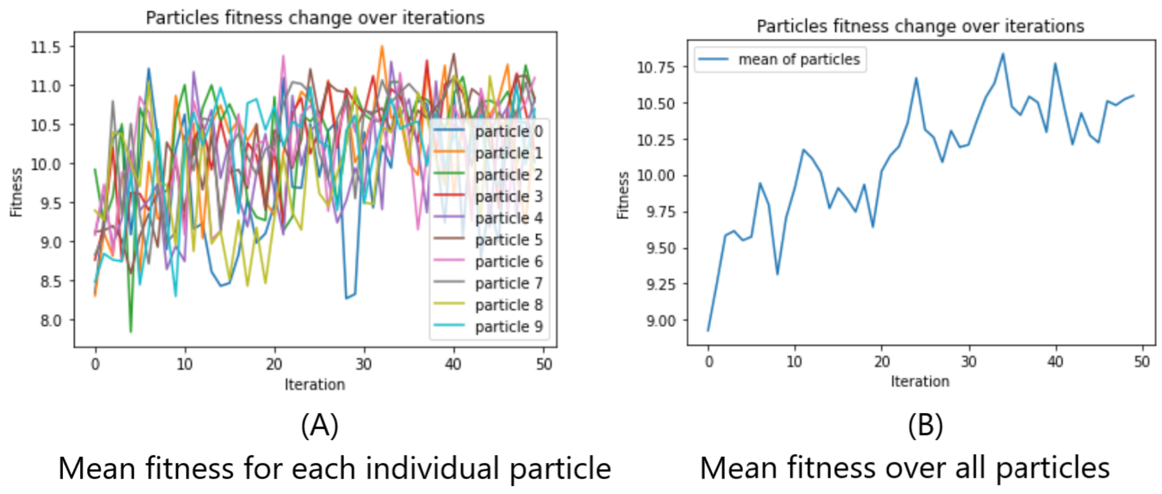

Figure 1: Learning PSO hyperparameters with PSO for the complex dataset.

2 Development Rationale

Our rationale was to create 2 submodules: ANNModel and PSO that could be used to build a fully connected neural network or perform PSO independently. We wanted the submodules to be completely decoupled to allow PSO to work on arbitrary Optimisation problems.

ANNModel is only able to create a fully connected neural network, we made this descision to simplify the process of vectorizing the parameters of the network for optimisation with PSO, this also made converting the vector back into a model easier.

The ANNModel's design was inspired by TensorFlow & Keras (Keras 2020; Tensorflow 2020), specifically when defining the shape of the neural network. For example you can instantiate the empty network, define the input and result vectors, the layers and then, finally, compile the model. Once compiled you can perform a single pass on the model with either random weights or activations, biases and weights defined by a vector.

The PSO class utilises a Particle class to abstract away some complexity. To use the PSO class define the hyperparameters in the constructor (as described in the documentation), and then specify the fitness function and search dimensions for PSO.

We created an interface for PSO Called Optimisable, any class that properly implements this interface can be used with our PSO implementation. The beauty of this technique is that it allowed us to implement this interface on our PSO class and construct a (PSO) optimiser for our PSO hyperparameters for a specific model shape, for clarity we refer to this outer PSO optimiser as "meta-PSO" and the inner PSO optimiser as "inner-PSO".

This interface also allowed us to create some wrapper classes(PSOHistory, PSOFittest) that can store detailed data about all the hyperparameter settings of the model being optimised. Figure 2 shows the definitions of the hyperparameters and boundaries that our meta PSO algorithm searches over.

Figure 2: A list describing the dimensions and boundaries that PSO will search over. See the documentation for more information (Fay-Hunt and Szymczak 2020).

```
def dimension_vec(self):
    swarm_size = (10, 100)
    informants = (4, 8)
    alpha = (0.01, 2.0)
    beta = (0.01, 2.0)
    gamma = (0.01, 2.0)
    delta = (0.01, 2.0)
    epsilon = (0.01, 2.0)
    return [swarm_size, informants, alpha, beta, gamma, delta, epsilon]
```

3 Testing Methodology

As explained in the previous section, we used a method to find good PSO hyperparameters by applying PSO to another PSO that itself tries to optimize our ANN for a defined dataset. This allowed us to investigate a wide search space of potential optimal hyperparameters for the PSO that was used to optimize the given ANN.

Testing steps:

1. Run the Meta_pso notebook with each dataset and record the resulting vectors.
2. Copy the vectors from Meta_pso to the evaluation notebook.
3. For each pair of datasets from Meta_pso set the corresponding training/testing data, the decimal places to compare, the PSO, and ANN parameters.
4. Run the tests and record the results.

4 Results

	Experiment	Data	Fitness	Loss	Score*
Cubic	Best ANN params	Train	4437	0	97%
		Test	6847	0	100%
	10 run mean from best PSO params	Train	47	0.028	12%
		Test	61	0.022	9%
Linear	Best ANN params	Train	3.005e+17	0	100%
		Test	3.380e+17	0	100%
	10 run mean from best PSO params	Train	223	0.033	31%
		Test	219	0.029	34%
Tanh	Best ANN params	Train	236154	0	100%
		Test	74997	0	100%
	10 run mean from best PSO params	Train	49	0.078	26%
		Test	55	0.101	22%
Sine	Best ANN params	Train	487.8	0.002	31%
		Test	459.3	0.002	39%
	10 run mean from best PSO params	Train	12.28	0.083	11%
		Test	12.02	0.086	8%
Complex	Best ANN params	Train	20.17	0.05	9%
		Test	16.05	0.062	18%
	10 run mean from best PSO params	Train	7.83	0.129	12%
		Test	20.96	0.049	17%
XOR	Best ANN params	Train	9096925098444960	0	100%
		Test	16.05	0	100%
	10 run mean from best PSO params	Train	299545	0.135	85%
		Test	263010	0.109	91%

Table 1: Table of results, showing the scores from the best ANN parameters from meta-PSO, and the mean best scores from an optimiser using the hyperparameters discovered in meta-PSO

5 Conclusion

5.1 Discussion

Flaws in testing:

- We use a fixed model structure for our neural network during testing, cannot generalise to other models.

- The fitness function we used was simply 1/loss, this was easy to implement but may have impacted the ability of PSO to escape local maximum. In future a linear fitness value may have been preferred.
- During meta-PSO we only took the mean of 10 inner-PSO runs to evaluate the fitness of each inner-PSO hyperparameter configuration. More would have been better, but impractical in terms of time.
- The best ANN hyperparameters we observed appeared to be a result of running at least 12.5 million different ANN hyperparameter configurations each time we ran meta-PSO in an attempt to optimise the inner-PSO hyperparameters for a given dataset.
- This might suggest the highest performing ANN hyperparameters discovered during our search for good PSO hyperparameters were more due to reinitialising PSO so many times than actually finding some really good PSO hyperparameters.
- We frequently observed PSO getting stuck in local maximum, this is reflected by the disparity in table 1 between the 10 run mean results and the results recorded from the best found during meta-PSO
- meta-PSO found a mean swarm size of 74 particles, this somewhat supports the observation that this is more of a brute force approach and (this implemenation) is very wasteful in terms of computation.
- Poor computational performance of our implementation (Memory access bottlenecks?)

5.2 Further work

From the experimental results we have observed we feel future experiments should look closer at other fitness evaluation techniques. An investigation into different hyperparameter search dimensions for both the ANN model and PSO would be interesting, some hyperparameters were found very quickly and it may be worth investigating if we could freeze those once they are found during optimiser runtime and reduce the search dimensions (careful consideration should be made with getting stuck in local maxima here).

A proper benchmark suite should be implemented to identify the serious bottlenecks in computation and memory access so we can begin to address them, and run more comprehensive experiments within a reasonable timeframe. Likewise a unit testing suite would be helpful in idientifying any remaining bugs in this codebase, and may also provide some insight into the performance bottlenecks.

Implementing PSO visualization tools to view how the hyperparameters are affecting the fitness in real-time would provide more insight into what is going on with particles during optimization. This information might allow us to identify hyperparameters that tend to result in the particle being caught in local maxima.

5.3 Finally

Findings in (Garcia-Nieto and Alba 2012; García-Nieto and Alba 2011) indicate that 6 to 8 informants is generally a good number of informants that each particle should have. Our own findings support this.

A References

References

Fay-Hunt, Sam and Kamil Szymczak (2020). *Biologically Inspired Computation Coursework Documentation.* URL: `https://www2.macs.hw.ac.uk/~sf52/Bio-Comp-docs/html/index.html` (visited on 11/22/2020).

Garcia-Nieto, José and Enrique Alba (July 7, 2012). "Why Six Informants Is Optimal in PSO". In: *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation.* GECCO '12. New York, NY, USA: Association for Computing Machinery, pp. 25–32. ISBN: 978-1-4503-1177-9. DOI: `10.1145/2330163.2330168`. URL: `https://doi.org/10.1145/2330163.2330168` (visited on 11/21/2020).

García-Nieto, José and Enrique Alba (Jan. 1, 2011). "Empirical Computation of the Quasi-Optimal Number of Informants in Particle Swarm Optimization". In: Genetic and Evolutionary Computation Conference, GECCO'11, pp. 147–154. DOI: `10.1145/2001576.2001597`.

Keras, team (2020). *Keras Documentation: The Model Class.* URL: `https://keras.io/api/models/model/#model-class` (visited on 11/22/2020).

Tensorflow (2020). *Module: Tf.Keras.Layers — TensorFlow Core v2.3.0.* URL: `https://www.tensorflow.org/api_docs/python/tf/keras/layers` (visited on 11/22/2020).