2024-2025

ANN-PSO Optimization

Coursework for Biologically Inspired Computation (F21BC)

BY:- HARIKA MAHALAKSHMI SRIDHAR (H00464781)
AND
SYED ARIF ALI (H00484788)

F20BC/F21BC - Coursework Group Signing Sheet

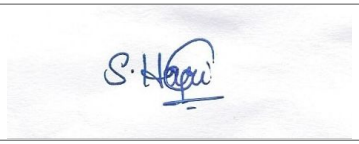
INSTRUCTIONS

Each group should print one copy of this sheet, fill it in, sign it and hand it in together with the CW.

The group is to agree the % of total effort put in by each of the group member's to the development of the coursework that should sum up to a 100%. Each member must sign to show they agree the % of total effort and the sheet must be handed in together with the CW.

No marks will be issued until we have this signed copy.

Group Number: PG15

PRINT NAME AND STUDENT ID NUMBER	SIGNATURE	CONTRIBUTION TO PROJECT	% TOTAL EFFORT
HARIKA MAHALAKSHMI SRIDHAR (H00464781)		PSO Implementation and Visualisation consisting of files - PSO.py, Particle.py, example.py. Implemented core PSO algorithm visualising particle behaviour and the error. Implemented features such as inertia decay, alpha decay, bounded velocities, informant type selection, set PSO parameters and linked the MLP with PSO with file mlp_with_pso.py. PSO documentation and Report.	50%
SYED ARIF ALI (H00484788)		MLP Implementation and layer implementation consisting of files - MLP.py, Layer.py, Activations.py, Cost.py, forward_pass.py. Defined the Activation functions, Cost functions, Layer initialisation and implemented forward propagation testing on the given dataset. MLP Documentation and Report	50%
Total:			100%

DECLARATION 1: We agree that the above information genuinely reflects the effort put in by group members and we **WISH** / **DO NOT WISH** (circulate as appropriate) marks to be allocated equally.

DECLARATION 2: We declare that the coursework has not been copied or plagiarised in any way.

NOTE: Unless indicated otherwise, all marks will be allocated equally. In case of any dispute on group contribution/mark allocation, the course leader will be final arbiter.

Please refer to the student handbook on notes on plagiarism. All cases of detected plagiarism will be reported to the disciplinary committee for consideration.

Table of Contents

Implementation

The code implements two core components:

1. **Artificial Neural Network (ANN):** A feedforward multilayer perceptron (MLP) designed for regression tasks. Configurable options includes the number of layers, number of neurons in each layer, and activation functions (ReLU, tanh, and logistic).
2. **Particle Swarm Optimization (PSO):** An optimization algorithm that updates a population of particles to minimize a fitness function. Each particle encoded ANN parameters (weights, biases, and optionally, activation functions).

The ANN and PSO were integrated such that PSO optimized the weights, biases, and activation functions of the ANN layers to solve a regression problem.

Overview of the Code Structure

How to run?

Run the mlp_with_pso.py file to run the MLP optimization using PSO. The MLP parameters are user defined, whereas, the PSO parameters must be configured via the pso_parameters.py file. To run just the forward pass of the MLP, the foward_pass.py must be run and the layer configurations must be set in it.

The implementation comprised the following files:

ANN Code:

1. **Activations.py:** Defined activation functions.
2. **Layer.py :** Represented individual ANN layers
3. **MLP.py :** Implemented the (MLP) multi-layer perceptron.
4. **Forward_pass.py :** Conducted initial forward pass experiments to test different architectures.

PSO Code:

1. **Particle.py:** Defined particles and their attributes.
2. **PSO_parameters.py:** Defined configurable PSO parameters.
3. **PSO.py:** Implemented the PSO algorithm.

Integration and Experiments:

1. **MLP_with_PSO.py**: Integrated PSO and ANN, allowing ANN training using PSO.

Design Decisions

I. **ANN Design**: A flexible architecture with configurable activation functions that allows testing the effect of architecture on performance.

II. **PSO Modifications**: Used informants instead of global best influence, decay functions for step size and inertia, and boundary handling for weights and activation function indices. Incorporated a reinitialization strategy for value(loss) stagnation avoidance.

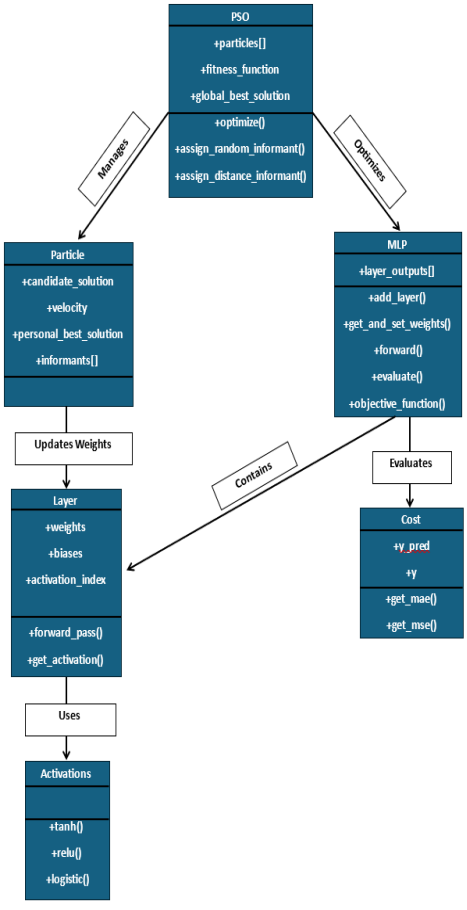


Fig. 1.1 –Code Structure

Code – Analysis

I. Layer.py

Key Components:

a. Initializes each layer's parameters like the activation function, weights, biases.

b. The get_activation() function handles the activation function selection process based on layer configuration and position. For regression tasks, the last layer returns Z without activation, while the hidden layers use PSO optimization to determine activation functions when not explicitly specified. The forward_pass() function then processes inputs through the linear function and chosen activation function.

II. MLP.py

Key Components:

a. The MLP class initializes the layer_outputs for each layer.

b. add_layer() function adds the layer(an array that includes weights, bias and activation for that layer) to the layer_outputs list.

c. get_and_set_weights() checks if the weights is equal to None(happens initially), if yes(means getting the weights), then create a flat(1-d array) weights array, and iteratively adds the layer's weights, bias and activation values to the flat_weights array.

d. In the case that the weights parameter is not None(weights updated in the objective_function() method), it is setting the weights for the layer.

e. The evaluate() function returns the cost(mae).

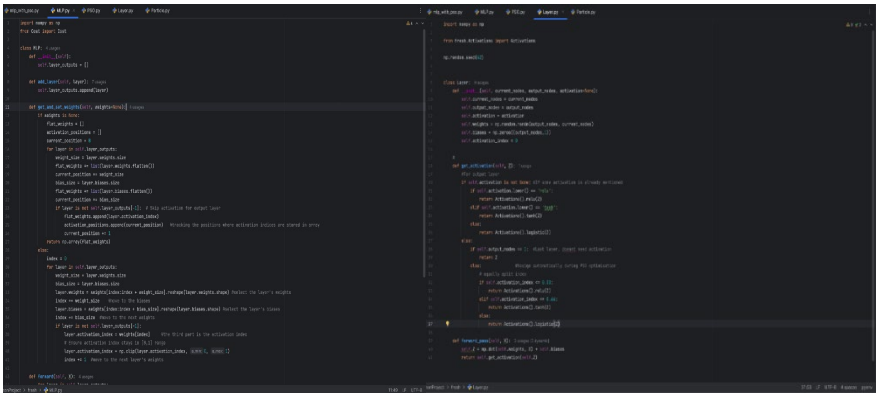


Fig. 1.2 – Code snippets of Layer.py and MLP.py

III. Forward_pass.py

The run_example() function processes the data by splitting it into features(X0 and target(y). Then, it is split into train and test sets. Following which, the MLP() object is created which is then used to add required layers to the ANN. The layer configurations are customisable, and the output is evaluated using the Mean Absolute Error function.

MLP forward pass testing run with the given dataset. Output values not optimized by any training. MAE obtained : 32.48379665 with architecture [(8, 16, 'logistic'), (16, 8, 'logistic'), (8, 1, None)]

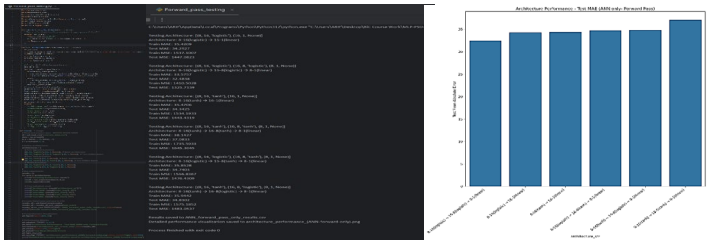


Fig. 1.3 – Forward pass testing Output and Plots

IV. Particle.py

Every particle in the PSO population is initialized as an Object of this class. Initially, the attributes candidate solution and velocity are set to random, personal best value and current values are set to infinity so they can capture new values properly, personal best position would be the same as current position. The informant list stores the informants for each particle. The dimensions will be the total number of parameters to be optimized by the PSO.

V. PSO.py

This class imports all the parameters for PSO from the pso_paramters.py file. The main objectives of this class are to optimize the fitness function by updating the particles, visualize two dimensions of all particles as they converge along with an updating plot of the MAE values w.r.t., the iterations.

Informant assignments happen based on the informant choice from user. For distance-based informants, the KNN approach is followed by calculating the **Euclidean distance** between the informant and the particle. The K nearest neighbors are appended as informants. Random informants are assigned from a random choice. The informants are updated once every few iterations as specified.

The optimise() function performs optimization of weights and activation index. One of the main features of this function is the **adaptive updates** to the PSO parameters alpha(inertia) and epsilon(step size).

The **velocity update** is done according to the formula:

new velocity = inertia_component + cognitive_component + social_component + global_component [Batatia,H, 2024].

```
new_velocity = (current_alpha * particle.velocity + beta * r1 * (particle.personal_best_solution -
particle.candidate_solution) + gamma * r2 * (best_informant.personal_best_solution - particle.candidate_solution) +
delta * r3 * (self.global_best_solution - particle.candidate_solution))
particle.velocity = new_velocity
#...some value clipping followed by updating the candidate solution/position
particle.candidate_solution += self.epsilon * particle.velocity
```

Fig. 1.4 – Code snippet of Velocity Updation

The global best values are tracked to make sure that the particles are not stuck at one loss value for too long. A strategy has been implemented to avoid stagnation. When stagnation persists for a reset threshold value, the particles are repositioned around best solution, velocities randomized within defined range and particles are spread within a certain range to escape local optimum. The weights are clipped to be in the range (-5, 5), whereas the activations indices are clipped to be between 0 and 1 to match the different activation thresholds set (<0.33 for relu, <0.66 for tanh, and <1 for logistic).The visualisation provides insights on the pattern of convergence and the MAE values as the number of iterations increases.

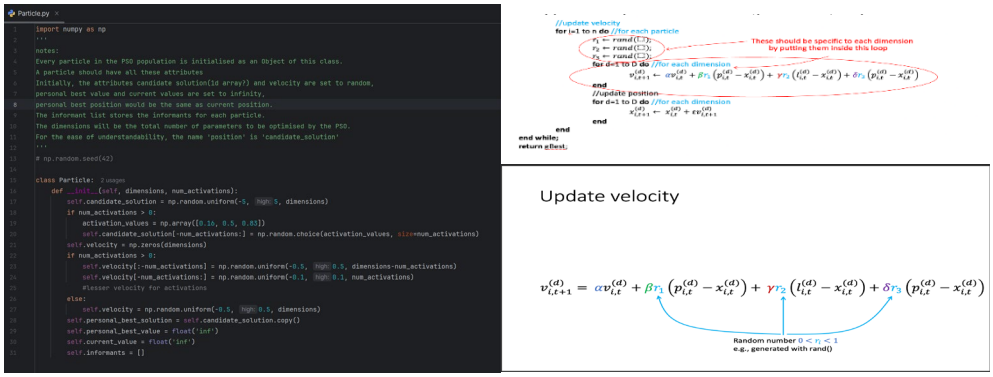


Fig. 1.5 – Code snippet of Particle.py and logic of Velocity updating

VI. **mlp_with_pso.py:**

This file is the starting point for execution of the ANN optimisation using PSO. The different parameters for the MLP are obtained from the user, along with the informant type, number of iterations, initial step size, number of particles, dataset path and the number of PSO particles. The layers are added to the MLP, which is then optimised by the PSO through iterative updating of weights and biases. A sample of the final output predictions are printed, and the test and train scores for MAE are also displayed. For advanced optimisation, the input data is scaled before further processing to remove any noise in the data.

Experiments and Results:

1. Questions Addressed

a. What effect does the ANN architecture (number of layers, neurons, and activation functions) have on performance (Forward pass only testing)

Several experiments were conducted with just the MLP's forward propagation(picking the minimum value obtained over 10 runs of the forward_pass.py program with an architecture), with no means to optimize its performance.

1. For one hidden layer configuration, the relu activation generally has a good performance and shows best value with 8 neurons. Tanh is the most stable across all sizes, and logistic has a variable performance. The difference between architectures are relatively small.
2. Network size doesn't significantly impact the performance, but 8 neurons seem to be a good balance of performance vs complexity.

Simple Architecture (1 hidden layer with 4 nodes)	Train MAE (mean± std.deviation)	Test MAE (mean± std.deviation)
Relu	36.19 ± 0.54	35.07 ± 0.50
Tanh	36.15 ± 0.07	35.10 ± 0.11
Logistic	36.27 ± 0.65	35.16 ± 0.65

Simple Architecture (1 hidden layer with 8 nodes)	Train MAE (mean± std.deviation)	Test MAE (mean± std.deviation)
Relu	35.87 ± 0.33	34.74 ± 0.36
Tanh	36.16 ± 0.03	35.04 ± 0.06
Logistic	36.29 ± 0.54	35.18 ± 0.53

On adding another layer, Simple Architecture (1 hidden layer with 16 nodes)	Train MAE (mean± std.deviation)	Test MAE (mean± std.deviation)
Relu	35.82 ± 0.97	34.68 ± 1.00
Tanh	36.14 ± 0.05	35.03 ± 0.05
Logistic	35.74 ± 0.68	34.61 ± 0.69

On experimenting with an additional layer with sizes 4, 8, and 16, it was observed that compared to a single layer, two layers are better. When the first layer is set to logistic activation, the results were optimized, but due to the output range limitation especially in cases like the concrete compressive strength prediction, it is safer to ignore logistic activations in the first layer. The next best option is relu for the first layer and logistic for the second layer, as relu in the first layer allows for unbounded positive values and logistic in the second layer helps compress the values between 0-1. To experiment with a third layer, keeping the final layer activation as relu would give proper output ranges for weights for the concrete compressive strength regression problem. In this case, the three layers tested were [8,4,4], [8,4,2], [8,4,8], it was observed that expansion to 8 doesn't help performance with this configuration. The best was found to be [8,4,4] with high performance (35.34 on train) and a lower variance. Thus the further experimentations will be conducted with [8(relu), 4(logistic), 4(relu)] architecture.

2. How should solution evaluations be allocated in PSO (e.g., swarm size vs. iterations)?
The number of solution evaluations obtained can be calculated as the number of particles in the swarm(swarm size) multiplied by the number of iterations. To keep computational complexity low, it might be necessary to keep them balanced.
But **Clerc, M., (2002)** recommends swarms with more than 100 particles for problems like neural networks.
Assuming constant values for all parameters, the observations recorded are:

Evaluations	Swarm Size	Iterations	Best Train MAE	Best Test MAE
15000	30	500	12.0567	11.841
15000	500	30	13.6032	13.55
30000	100	300	12.5425	12.748
30000	300	100	12.7572	13.68
30000	30	1000	12.6989	12.795
500000	100	5000	9.6383	10.0915
500000	200	2500	10.0340	10.877

Based on the experiments conducted, small swarms with more iterations perform better than large swarms with fewer iterations. But when the number of iterations is increased, there is more time for the swarm to stabilize. In this case, 5000 and 2500 iterations are providing the best values (9.63 and 10.03 respectively). But sometimes, with more iterations, in the chance that the particles find a local optimum, there is some stagnation at the position. Further, it is seen that on many occasions there are only minor improvements in the value after convergence. According to (**Shi, Y., 1998)**, 20-30 particles maybe enough for basic optimization. But if the swarm size is small, like 30 particles here, then a greater number of iterations(500) are needed for fine-tuning.
The reset mechanism approach has been implemented in the PSO file in order to tackle this problem of balancing swarm size and particles. This mechanism allows for the swarm size to be smaller, while the iterations can be larger, and after every few iterations, the particles are repositioned around the global best. Similarly, the distance-based informants benefit from larger swarms. On the contrary, strategies like the parameter decay perform better with a greater number of iterations, or even with balanced swarm and iterations. Thus further experiments will be carried out with 100 particles and 5000 iterations.

a. What is the effect of varying acceleration coefficients in PSO?
The Beta and Gamma values are experimented with and they produce the results:

Test type	Beta	Gamma	Delta	Train MAE	Test MAE
High Social	1.0	2.5	1.5	9.3471	9.3561
Low Social	1.5	0.5	1.5	10.06	10.305
High Cognitive	2.5	1.0	1.5	9.6035	9.7612
Balanced	1.5	1.5	1.5	9.5643	9.683

Based on the paper by (**Zhan et al. ,2009)**, when delta is kept at 1.5, it provided better performance than δ=1.0, particularly for high-dimensional problems like neural network optimization.

The inertia(alpha) is set to decay from 0.9 to 0.4 for gradual transition, thus facilitating more exploration in the beginning and resort to more exploitation at the end for better convergence. Similarly, for the step size, higher values help find a minimum and further decay delays convergence to avoid local minima. It is observed

that instead of having fixed values for alpha, decay provides better results as the exploration and exploitation are balanced, according to (**Shi, Y. ,1999).**

When the social component (Gamma) is high, the particles seem to converge quickly to a local minimum. This is because the influence of particles on each other is too much. But when the cognitive component is higher than the social component, the particles struggle to converge and spend much time exploring the space. This approach seems to demand more number of iterations for it to converge, but will prevent local minimum.

b. What is the effect of varying the number of informants?
So, the system is tested with 4, 6, and 8 informants over 3 runs to account for the stochasticity of the PSO. The best performance was achieved with 6 informants, as per **Garcia-Nieto (2012)**.

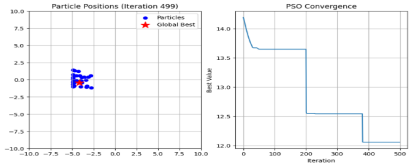


Fig1.6 Swarm size/Iteration: 30, 500

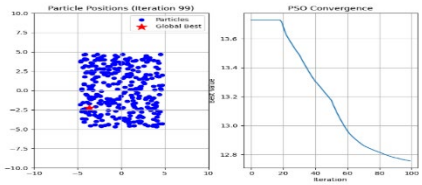


Fig1.7 Swarm size/Iteration: 100, 300

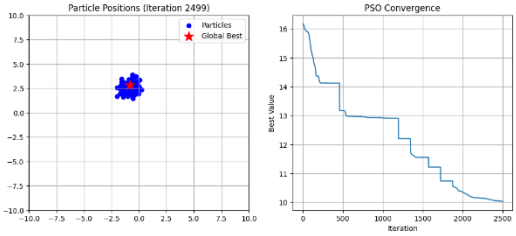


Fig1.8 Swarm size/Iteration: 200,2500

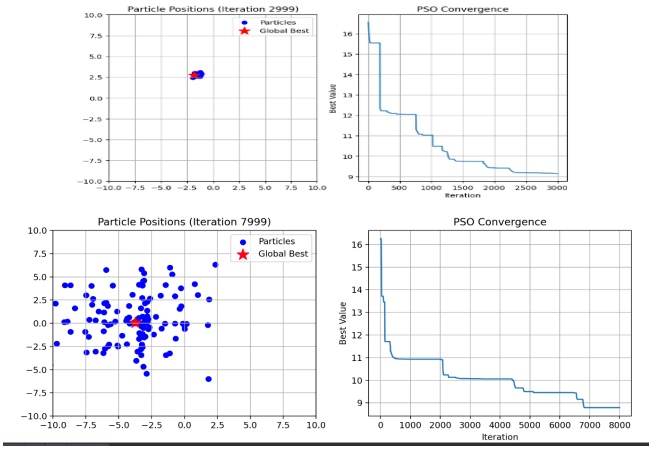


Fig1.9 Plots over different iteration ranges.

Discussion and Conclusions:

It is observed that the most optimal configuration of the ANN architecture for the concrete compressive strength dataset is 3 layers with 8 nodes in the first layer, 4 in the second and 4 in the third layer, with 1 output node in the last layer. From the initial experiments it seems that ReLU combined with Logistic and ReLU gives the most optimum values. This must be because initial ReLU helps with feature extraction, middle Logistic provides bounded outputs, and final ReLU maintains positive outputs, which is essential here, as the target value cannot be negative. As far as the PSO coefficients are concerned, a balance between the cognitive and social components is crucial. The best cognitive weight is 1.5, and the best social weight is 1.5. Since inertia decay is implemented, it gradually decreases from 0.9 to 0.4. Similarly, the step size decrease with every iteration, encouraging exploration in the beginning and exploitation by the end. By having two informant types specified, many valuable insights can be discovered. In this case, since the number of iterations is high, if the distance-based informants are chosen, then it increases the computational complexity. Especially in neural networks, since there are many dimensions, the Euclidean distance method may not perform the best. With these values, the Multi-Layer Perceptron can produce outputs with a loss in the range 7-13.

The research has successfully experimented with different layer architectures for the ANN and parameters for the PSO to produce the optimal values to efficiently improve the ANN's performance on the concrete compressive strength dataset provided.

References:

Batatia, H. , 2024 , Week 7 Lectures - PSO Algorithms , Canvas , Slide-16 and Slide 17

Clerc, M. and Kennedy, J., 2002. The particle swarm-explosion, stability, and convergence in a multidimensional complex space. *IEEE transactions on Evolutionary Computation*, *6*(1), pp.58-73.

Kennedy, J. and Mendes, R., 2002, May. Population structure and particle swarm performance. In *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600)* (Vol. 2, pp. 1671-1676). IEEE.

Shi, Y. and Eberhart, R., 1998, May. A modified particle swarm optimizer. In 1998 IEEE international conference on evolutionary computation proceedings. IEEE world congress on computational intelligence (Cat. No. 98TH8360) (pp. 69-73). IEEE.

Van den Bergh, F. and Engelbrecht, A.P., 2010. A convergence proof for the particle swarm optimiser. *Fundamenta Informaticae*, *105*(4), pp.341-374.

Zhan, Z.H., Zhang, J., Li, Y. and Chung, H.S.H., 2009. Adaptive particle swarm optimization. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, *39*(6), pp.1362-1381.