TNSP Solution Based on PSO-NN Using CPP

Report of Computational Intelligence Course

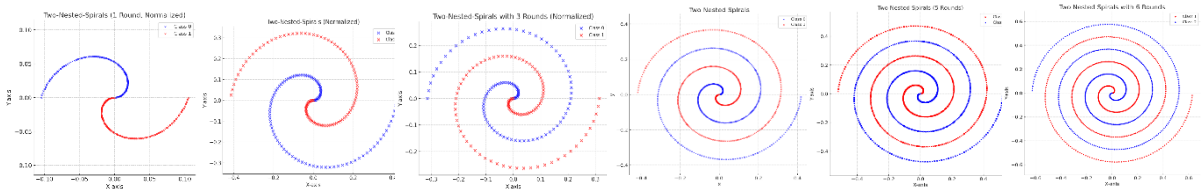
LU Weicheng (Student ID: 44241645)

1. Overwiew

The Two-Nested-Spirals Problem (TNSP) is a well-known benchmark problem in the field of machine learning and neural networks. It involves classifying points arranged in two intertwined spirals into two distinct classes. The task poses significant challenges due to its highly nonlinear nature and the intricate relationship between data points. TNSP is often used to evaluate the ability of machine learning algorithms, particularly neural networks, to model complex decision boundaries. Success in solving TNSP demonstrates the algorithm's potential for handling intricate classification problems in real-world applications.

To slove TNSP, instead of using the conventional 'gradient descent' algorithm, I utilize PSO algorithm to train the weights W and biases b of the neural network. Also, I use Cpp other than Python. PSO has good global search capabilities. By adjusting the parameters (w,$c1$,$c2$), it can make the neural network converging to the global optimum theoretically, other Sthan local optimum.

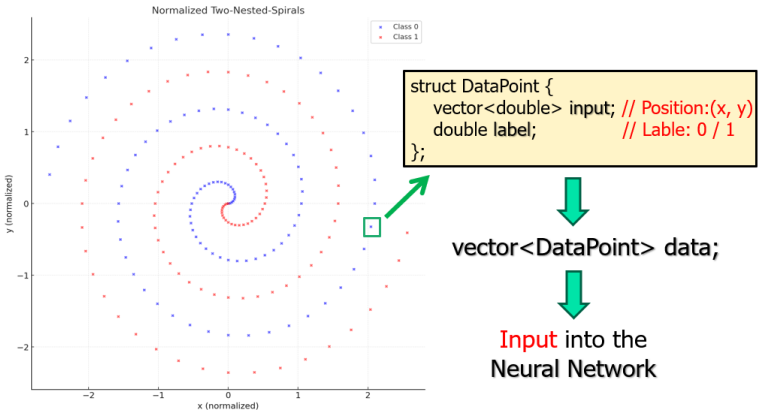
Starting with a single-turn double spiral, the number of turns was gradually increased up to a maximum of six turns, and the performance of the PSO-NN model on the TNSP was trained and tested at each step. It was observed that as the number of spiral turns increased, and the training dataset size grew proportionally, the convergence speed of the PSO-NN model significantly decreased. However, the final classification performance remained excellent, achieving a classification accuracy of 98% or higher on the validation dataset in most cases.



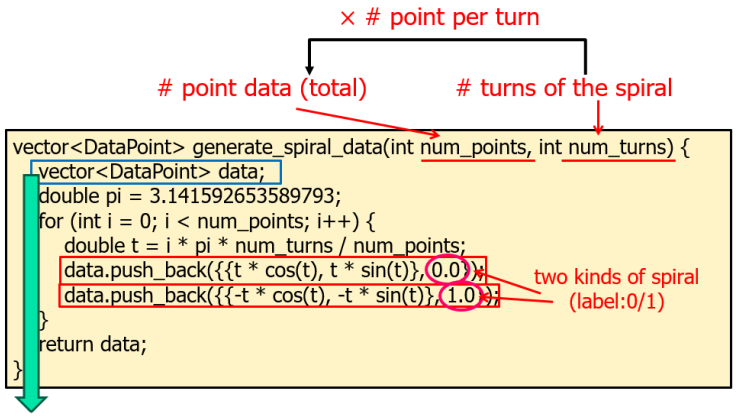
2. Generation of Spirals Data

To implement PSO-NN using C++ without relying on frameworks like TensorFlow, I will build the entire algorithm and neural network from scratch. For the double spiral data, each point on the spiral image (each data point) is stored as a **struct DataPoint**. The structure contains the coordinates of the point **(x, y)** and its classification **label** (0 or 1). When each data point is fed into the PSO-NN for training, the coordinates (x, y) are used as the input. The classification output is then compared with its label, and the PSO algorithm is applied to perform "backpropagation," optimizing the weights and

biases within the neural network.

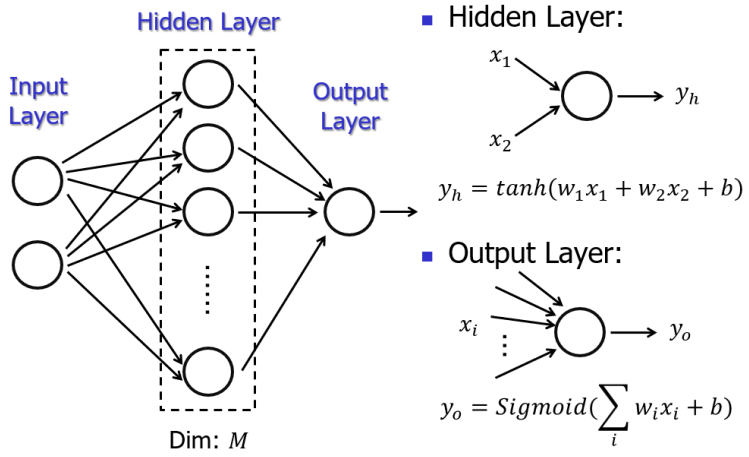


In the **generate_spiral_data** function, a list **vector<DataPoint> data** is created to store all the points. The parameters **num_points** and **num_turns** are used to control the amount of data generated and the number of spiral turns. There is a multiplicative relationship between **num_points** and **num_turns**: $num\_points = num\_turns * num\_points\_per\_turn$. This ensures that as the number of spiral turns increases, the amount of training and validation data generated also increases, thereby maintaining the accuracy of the model. The dataset fed into the PSO-NN is the collection **vector<DataPoint> data**.

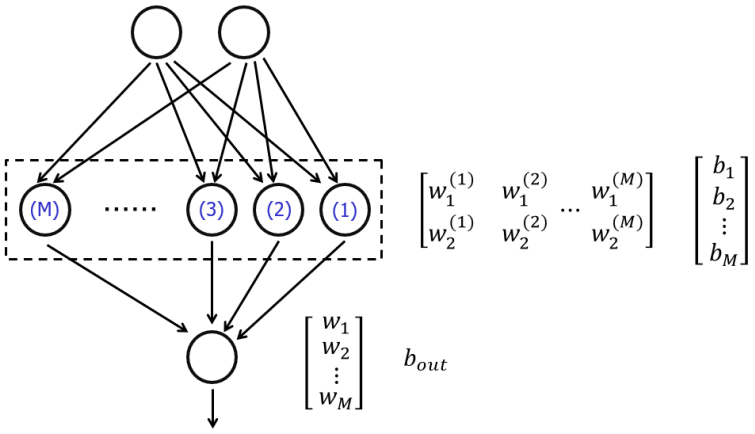


3. Design of 2-layer Neural Network

The project requires solving the TNSP using a two-layer neural network, consisting of an input layer, a hidden layer, and an output layer. The input layer has two nodes, which take the coordinates x and y of the double spiral data as input. In the hidden layer, I used a basic linear fitting function $y = w_1x_1 + w_2x_2 + b$ and selected the activation function as $tanh$. In the output layer, I also applied a basic linear fitting function to accept N outputs from the hidden layer: $y = \sum_i w_i x_i + b$, followed by the activation function $Sigmoid$. Finally, the classification result (0 or 1) is determined by checking whether the output of the output layer is greater than 0.5.



The hidden layer and output layer include the **weights** and **bias** parameters of the neural network. The size of the parameter matrices changes with the number of nodes in the hidden layer. The basic structure is outlined as follows:



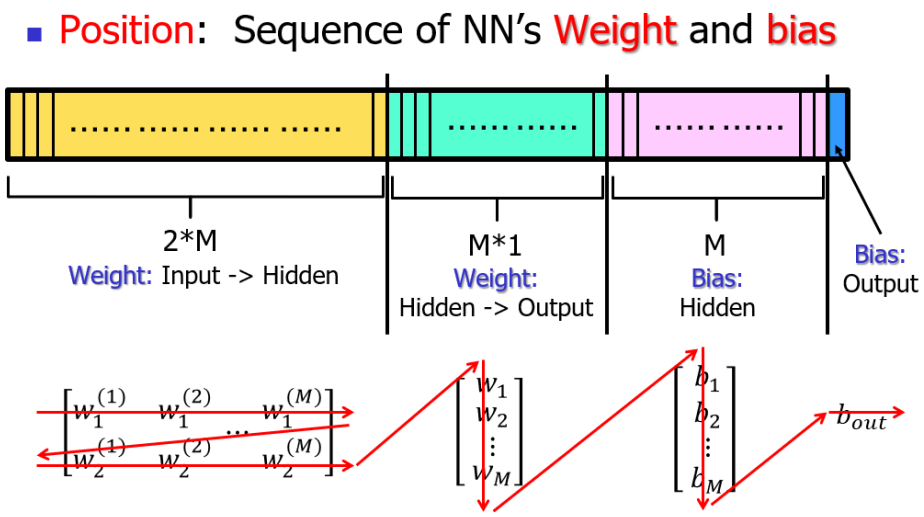
4. Design of PSO Algorithm

In PSO-NN, the PSO algorithm is responsible for optimizing the neural network parameters (weights and biases). In this project, the basic unit of the PSO algorithm, the "particle," is defined as a structure **struct Particle**, which stores key attributes such as **position**, **velocity**, **best_position**, and **best_fitness**.

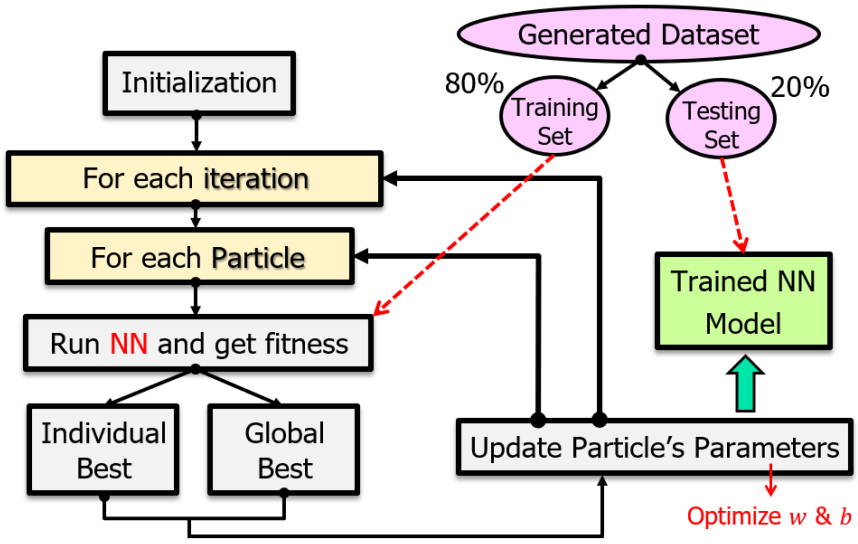
```
struct Particle {
    vector<double> position;
    vector<double> velocity;
    vector<double> best_position;
    double best_fitness;
};
```

Among these, the most critical attribute is the particle's position, which I designed to represent the collection of neural network parameters. Specifically, the parameters are flattened and stored in position in the following order: "hidden layer weights, output layer weights, hidden layer biases, and output layer biases." This approach ensures that each particle represents a neural network during the optimization process. As particles

move within the parameter space, the weights and biases of the neural network are effectively updated. This process is equivalent to training and optimizing the neural network parameters through the PSO algorithm.



This flowchart illustrates the training and validation process of the PSO-NN algorithm for solving TNSP. The PSO algorithm is first initialized, where the particles' positions and velocities are randomly initialized. During the training phase, for each iteration, every particle evaluates its performance by running the neural network using its current parameters on the training set and computing a fitness score. Each particle updates its individual best performance, while the swarm collectively identifies the global best particle. Based on these best positions, the particles update their positions and velocities. This process continues iteratively until maximum number of iterations is reached. After training, the global best particle represents the optimized neural network model, which is then validated on the testing dataset to evaluate its classification accuracy and generalization. The flowchart highlights the seamless integration of PSO to optimize neural network parameters without relying on traditional backpropagation, ensuring robust and efficient training while maintaining an objective evaluation on unseen data.



5. Train and Test Result

(1) Hyper-Parameters:

PSO-Parameter	**Value**
w	0.2
c_1	1.0
c_2	2.0
r_1	Random (0.0,1.0)
r_2	Random (0.0,1.0)
num_particles	1000
NN-Parameter	**Value**
input dim	2
# hidden_nodes	32
output dim	1
Data-Parameter	**Value**
validation_ratio	0.2

(2) Train & Test Result:

number of spiral's turns	**Amout of data**	**PSO-NN converge iteration time**	**Training time consumption**	**Final Verification Accuracy**
1	100	58	1min	100%
2	200	250	2min	99.8%
3	300	500	15min	>99%
4	400	2000	1h	>99%
5	500	5000	>3h	>98%
6	600	10000	>10h	>98%

If spiral data is easy: PSO-NN performs excellent. PSO-NN can converge very fast (within 50 iteration). If data becomes complex: PSO-NN converge slow. But at least, after long waiting the model can finally converge to a good situation, other than diverge or overfitting, which proves PSO-NN's success!

6. Other things deserves to talk

(1) During the debugging of the PSO-NN, I also tried different **nonlinear fitting functions** (like $y = w_1 sinx_1 + w_2 cosx_2 + b$), and different **activation functions** for hidden layer (like ReLU). However, finally it comes out that $y = w_1 x_1 + w_2 x_2 + b$ and $tanh$ performs best. The simplest is the best, maybe.

(2) During the debugging of the PSO-NN, I also tried to increase **the number of**

hidden layer's node in order to make it performs better. However, it comes out that too much hidden nodes will result in overfitting, also increase the training time. After many many tries, PSO-NN performs better in 32 hidden-nodes.

(3) During the debugging of the PSO-NN, I find that **the amount of training data** highly influence the training result of the model. If the amount of training data keeps small when predicting complex spirals (like 5,6 rounds), the model will never be trained well, and the testing accuracy will keep in a low stage (about 60%).