

Title: Application of different optimization technique on “Thyroid Detection” dataset for supervised learning (classification).

Abstract

In this experiment, different optimization techniques like “Genetic Algorithm” (GA) and “Particle Swarm Optimization” (PSO) were implemented from scratch to find the best set of weights to classify “Thyroid Detection” dataset. Different techniques were compared in terms of classification accuracy, mean squared error and categorical cross entropy. Without detailed hyperparameters tuning, both optimizers score very closely while PSO seems like a much intuitive and efficient option.

Dataset

This dataset has a total number of 215 rows with 6 columns. Its column names are “CLASS”, “T3RESIN”, “THYROXIN”, “THYRONINE”, “THYROID”, “TST_VALUE”. “CLASS” is the column that contains the labels. The data can be classed into 3 classes, class “1” has 150 records, class “2” has 35 records and class “3” has 30 records. These data were separated into five sub sets, each contains train and test data records.

Preprocessing

Features of each sub set were scaled using **standard scaler** (implemented from scratch). Mean and standard deviation of each column were derived from the train set and used on both train and test set. Labels of each sub set were **one hot** encoded.

Neural Network and Controlled Variable

All optimizers were tested with the same neural network configuration. Our neural network model has only 1 hidden layer (4 neurons) and 1 output layer (3 neurons) and weights and bias were clipped within $[-3, 3]$.

Genetic Algorithm (GA)

Genetic Algorithm were proposed by John Holland in 1975 (Holland, 1975). GA adopt the metaphor of *species evolution* and *natural selection*. GA has a number of important phases that need careful design.

Encoding

Fundamentally, encoding phase is one of the most import phase. Since we are finding the *best weights* to be loaded to artificial neural network, we need a genetic sequence as long as the number of weights required by the neural network model. For our case, each stretch of genetic sequence is of length L .

$$L = \sum (\text{layer_i_input} * \text{layer_i_output} + \text{layer_i_output}) \{ i \text{ from } 0 \text{ to } N-1 \}$$

N is the total number of layer

Each component is of type *float32* and initialized from *uniform distribution* rather than normal distribution and constant value. I believe spreading the distribution wide a part could allow the model to explore a wider range of possibilities. Each component is treated equally and a set of *weight constraint* were applied to prevent the weights from over growing.

Fitness Evaluation

Since we are looking the best weights for our neural network, we defined the best weights as the weight combination that would produce the *lowest categorical cross entropy error (CCE)*. Weight sequence will be loaded into the model to get the loss value. Assume we have m number of genetic sequences, we have to load it into the model m times and generate m loss value. The computational complexity raised with respect to m .

```

1 def CCE(y_pred, y_true, em = None): # em were included for the sake of consistency
2     total_CE = 0
3     prob = []
4     for idx in range(len(y_pred)):
5         true_units = y_true[idx]
6         pred_units = y_pred[idx]
7         for i_target, i_pred in zip(true_units, pred_units):
8             if i_target == 1:
9                 # required a proper smoothing operation
10                i_pred = i_pred if i_pred > 1e-7 else 1e-7
11                i_pred = i_pred if i_pred < 1 - 1e-7 else 1 - 1e-7
12                prob.append(i_pred)
13
14    prob = np.array(prob, dtype='float32')
15    prob_tensor = tf.constant(prob)
16    log_tensor = tf.math.log(prob_tensor)
17    loss = tf.reduce_sum(log_tensor).numpy()
18
19    total_CE = -1 * loss / len(y_pred)
20    return total_CE

```

Picture 1. Implementation of CCE.

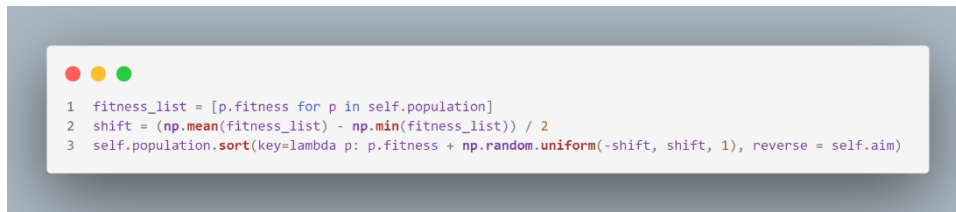
Selection

At this phase, we are expecting more than or equal to m number of genetic sequences in the population as the result of crossover and mutation phase. Selection phase could be further broken down into two sub phase which is **sampling** phase and **actual selection** phase.

Sampling

Sampling of sequence only based on their fitness could pushed us down to the trap of typical greed search. It's believed that certain less promising action could bring us greater benefits in near future. Therefore, these sequences were sort to **ascending order** according to their fitness with a **random shift**. The random shift is added to inject some randomness to the sampling process. At the end of this phase, only m number of sequences are left in the population.

Below I propose a simple way to calculate an **adaptive shift value**. Shift is to be half of the difference of **mean fitness** and **minimum fitness**. Random shift is to range between $-shift$ and $+shift$. This approach could allow the less appealing sequence to be selected while still maintained the likeliness of best sequence being selected.



```

1 fitness_list = [p.fitness for p in self.population]
2 shift = (np.mean(fitness_list) - np.min(fitness_list)) / 2
3 self.population.sort(key=lambda p: p.fitness + np.random.uniform(-shift, shift, 1), reverse = self.aim)

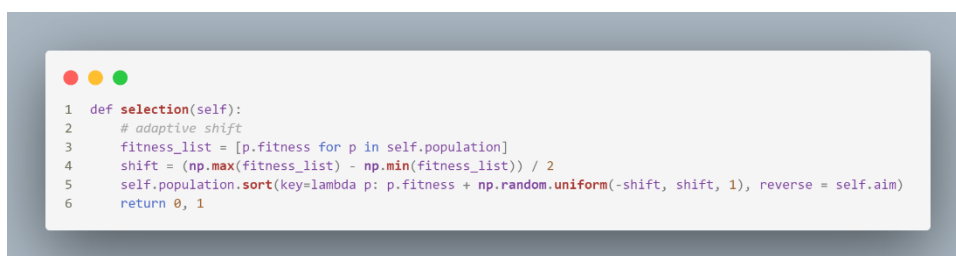
```

Picture 2. Implementation of adaptive shift.

As the nature of crossover, it's possible to have **identical** genetic sequence in the population. However, checking the similarity of each sequences through point to point matching could add quadratic computation cost to the algorithm. Hashing does reduce certain level of computation but computing of large floating value is great computational cost. Below I propose a less accurate but efficient heuristic approach to exclude the possibly **duplicated** sequence. First compute the **sum of weight** of each genetic sequence. Only the one sequence with the same sum will be copied into the new population. If the new population has less than m number of genetic sequences, **new gene** will be randomly generated to fill up the gap.

Actual Selection

In this phase, two sequence were selected to participate in the crossover phase. Each sequence deserved the possibility to be selected. Just like the sampling phase, the population were sorted with respect to the gene's fitness and a random shift. In order to prevent the chances of choosing two identical sequence for crossover, this function will always return **index 0 and 1** instead of random picking (with respect to the fitness). For this to work out, I modified the calculation of shift value slightly to ensure the **worst** sequence has the possibility to be selected. Instead of the mean of fitness, I choose to use the **max fitness**. With this I could map the gap between the worse and the best sequence.



```

1 def selection(self):
2     # adaptive shift
3     fitness_list = [p.fitness for p in self.population]
4     shift = (np.max(fitness_list) - np.min(fitness_list)) / 2
5     self.population.sort(key=lambda p: p.fitness + np.random.uniform(-shift, shift, 1), reverse = self.aim)
6     return 0, 1

```

Picture 3. Implementation of Selection.

Crossover

In this phase, two sequence meet together to reproduce a new sequence through **sequence swapping**. There are many ways to achieve this goal, single point, two point and uniform swapping.

Single-Point

Two sequence swaps at **one** randomly generated or given point to produce child sequence.



```
1 p1 = int(self.seq_len * np.random.uniform(0,1,1))% self.seq_len
2 child = copy.deepcopy(seq1)[:p1]
3 child = np.append(child, copy.deepcopy(seq2)[p1:])
```

Picture 4. Implementation of Single-Point technique.

Two-Point

Genetic sequence is separated into three sections by **two** randomly generated index *cut_1* and *cut_2*. The section between *cut_1* and *cut_2* are copied from sequence two and section out of *cut_1* and *cut_2* are contributed by sequence one.

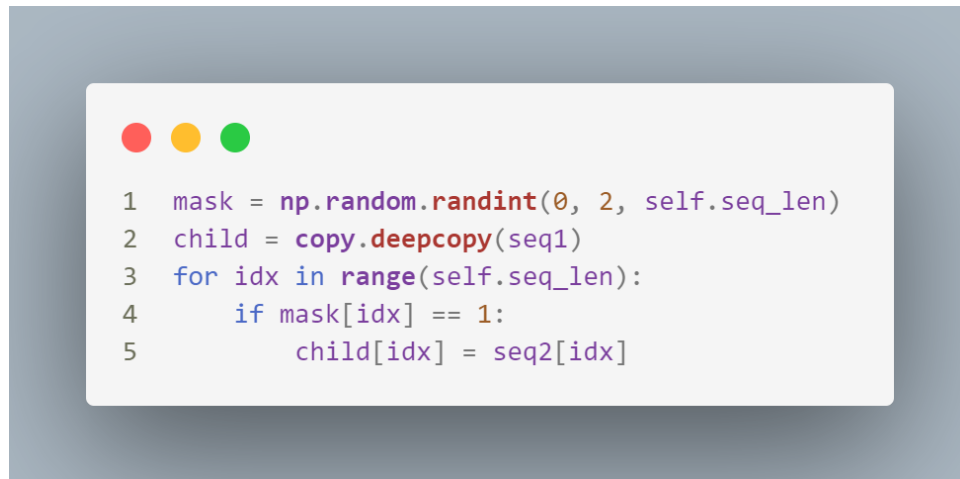


```
1 cut_1, cut_2 = int(self.seq_len * np.random.uniform(0,1,1))% self.seq_len, int(self.seq_len * np.random.uniform(0,1,1)) % self.seq_len
2 while cut_1 >= cut_2:
3     cut_1, cut_2 = int(self.seq_len * np.random.uniform(0,1,1))% self.seq_len, int(self.seq_len * np.random.uniform(0,1,1)) % self.seq_len
4     child = copy.deepcopy(seq1[:cut_1])
5     child = np.append(child, copy.deepcopy(seq2[cut_1:cut_2]))
6     child = np.append(child, copy.deepcopy(seq1[cut_2:]))
```

Picture 5. Implementation of Two-Point technique.

Uniform

A **mask** sequence of equal length was randomly initialized with **zero** and **one**. Two sequences exchange genetic information according to the mask sequence. Points with zero are copied from sequence one and points with one are copied from sequence two.



Picture 6. Implementation of Uniform technique

Mutation

In this phase, a sequence is selected without considering its fitness. Mutation phase is particularly important because it introduce **new possibility** to the optimizer which could help the optimizer to get out from the **local minimum**. N number of points will be mutated, these points were randomly selected **without repeat**. Hyperparameter *chaotic_rate* are introduced here, it's a rate sequence weight to be mutated. This can be tuned to optimize but set to constant 0.05 as a controlled variable throughout our research.

$$N = \text{sequence_length} * \text{chaotic_rate}$$
$$\text{rate} = [0, 1]$$

Two more hyperparameters are introduced here which are *radioactive rating* and *grow factor*. These selected points will be deducted by a random portion of radioactive rating and added by a portion of grow factor. Both hyperparameters should be a **sufficiently small** value to allow a more thorough **exploration**. They should **scale** down as the training progress.

```

1 def mutate(self, seq, rate = 0.05):
2     alien_seq = copy.deepcopy(seq)
3     n_points = np.min([int(len(seq) * rate), 1])
4     point_set = set() # prevent mutating the same point
5     for _ in range(n_points):
6         point = int(self.seq_len * np.random.uniform(0,1,1)) % self.seq_len
7         while point in point_set:
8             point = int(self.seq_len * np.random.uniform(0,1,1)) % self.seq_len
9         point_set.add(point)
10        new_code = alien_seq[point] - (self.rr * np.random.uniform(0,1,1)) + (self.gf * np.random.uniform(0,1,1))
11        alien_seq[point] = self.clip(new_code, self.weight_constraint)
12    return alien_seq

```

Picture 7. Implementation of Mutation.

Hyperparameter Scaling

A couple of hyperparameters can be scaled up or down throughout the training process. Those are *crossover rate*, *mutation rate*, *radioactive rating*, *grow factor* and *chaotic rate*. *Chaotic rate* is a floating point ranging from 0 to 1. During each iteration, at most $2 * m * \text{chaotic rate}$ number of events are expected. For the sake of simplicity, *chaotic rate* is set to a constant value 0.05.

Below I propose a scaling scheduler which execute every 100 iteration. The intention of this policy is to reduce the adjustment range to prevent over adjustment. However, mutation rate increases gradually with the decrease in radioactive rating and grow factor to help the model search within the nearby space.

```

1 self.rr = self.radioactive_rating[1] - (self.radioactive_rating[1] - self.radioactive_rating[0]) / max_iter * (max_iter - itr)
2 self.gf = self.grow_factor[1] - (self.grow_factor[1] - self.grow_factor[0]) / max_iter * (max_iter - itr)
3 self.crossover_rate = self.x_rate[1] - (self.x_rate[1] - self.x_rate[0]) / max_iter * (max_iter - itr)
4 self.mutation_rate = self.m_rate[0] + (self.m_rate[1] - self.m_rate[0]) / max_iter * (max_iter - itr)

```

Picture 8. Implementation of hyperparameter scaling.

Particle Swarm Optimizer (PSO)

PSO is an optimization technique that involve only simple computation (**non-statistical** and **doesn't required the problem be differentiable**). In PSO, candidate solutions which known as **particles** were tuned / move **iteratively** towards the best position according to the fitness function provided. On top of that, particles were tuned / fly towards its best-known position (**local**) and the best location known by the population (**global**). While the genetic sequences of GA are stateless, particles of PSO are **stateful** with the introduction of hyperparameters w and variable *velocity* which will be further explained in sections below. However, PSO does not promised a globally optimum solution because things get tricky when the best solution are **too far away** from the initial search space. Besides that, PSO has **more hyperparameters** than GA and everyone are highly influential.

Even though PSO has more hyperparameters than GA, its implementation is very simple. The most important phase of PSO is **vector update** and its implementation can be expressed in a few simple mathematical expressions.

$$\text{Equation 1: } v_{i_{t+1}}^s = w * v_{i_t}^s + r_{l_t}^s * c_l * (p_{i_l}^s - p_{i_t}^s) + r_{g_t}^s * c_g * (p_{i_g}^s - p_{i_t}^s)$$

$$\text{Equation 2: } p_{i_{t+1}}^s = p_{i_t}^s + v_{i_{t+1}}^s$$

In order to improve the efficiency of the optimizer and better fit my use case, below I propose a modification to both *Equation 1* and *Equation 2* in *Equation 3* and *Equation 4* consecutively. As mentioned $r_{l_t}^s$ and $r_{g_t}^s$ are random number range between $[0, 1]$ applied to add randomness so that a random portion of **local** and **global** adjustment are added to the adjustment of current time step t . In *Equation 3* I replace $r_{l_t}^s$ and $r_{g_t}^s$ with $r_{i_t}^s$ and $(1 - r_{i_t}^s)$. This modification not only reduce the **computational cost** but also preserve the original design concept.

$$\text{Equation 3: } v_{i_{t+1}}^s = w * v_{i_t}^s + r_{i_t}^s * c_l * (p_{i_l}^s - p_{i_t}^s) + (1 - r_{i_t}^s) * c_g * (p_{i_g}^s - p_{i_t}^s)$$

Based on *equation 1* and *2*, the particle with global best solution won't travel until its position is overtook by other particles. As a result, we could have a particle that is not moving. Below I modified *equation 2* to ensure there is a minimum charge(non-adaptive) in every update. $charge_{i_t}^s$ is a very small random number range between $[-alpha, alpha]$.

On top of these modification, I would like to propose a concept observed in every day to be applied to the **vector update expression**. Take human like us as an example, not every movement change all three **axes** of us. Every weight in particle sequence can be viewed as a **dimension**, every change in each dimension can have great impact to the particle's **fitness**. Therefore, I propose to keep some of the dimension **static** in some iteration. $mask_{i_t}^s$ is the random variable of either 0 or 1, product of $mask_{i_{t+1}}^s$ and $(v_{i_{t+1}}^s + charge_{i_t}^s)$ decide the adjustment of the current time step t . Equation 4 can be simplified by simple **if-else statement** rather actually computing the entire output when $mask_{i_t}^s$ is 0. The actual implementation can be found in the code attached. The original PSO and modified PSO will be called PSO and mask-PSO respectively throughout the rest of this report.

$$\text{Equation 4: } p_{i_{t+1}}^s = p_{i_t}^s + (v_{i_{t+1}}^s + charge_{i_t}^s) * mask_{i_t}^s$$

Experiment

Experiments were conducted to search for the most suitable configuration that perform well in this dataset. Some hyperparameters were set to constant throughout the research. All experiments on GA, weight constraint = [-3.0, 3.0), radioactive rating = 0.1, grow factor = 0.1, and population size $m = 50$ unless stated differently. All experiments on PSO, weight constraint = [-3.0, 3.0), velocity constraint = [-0.1, 0.1), and population size $m = 50$ unless stated differently.

GA

Crossover Techniques

Technique	G1	G2	G3	G4	G5	Average Accuracy
Single	0.9506	0.9568	0.9383	0.9568	0.9753	0.95556
Two	0.9444	0.9383	0.9444	0.9506	0.9259	0.94072
Uniform	0.8642	0.858	0.8272	0.833	0.8272	0.84198

Table 1. Average accuracy at epoch 500. Highlighted is the best performing technique.

Technique	G1	G2	G3	G4	G5	Average MSE
Single	0.1447	0.1583	0.16	0.1091	0.1216	0.13874

Two	0.137	0.1374	0.1551	0.1395	0.1439	0.14258
Uniform	2.8501	2.819	2.9758	2.9233	2.9175	2.89714

Table 2. Average MSE at epoch 500. Highlighted is the best performing technique.

Technique	G1	G2	G3	G4	G5	Average CCE
Single	0.3133	0.307	0.3023	0.2165	0.2739	0.2826
Two	0.2893	0.2534	0.3006	0.293	0.283	0.28386
Uniform	0.583	0.574	0.6728	0.6907	0.6791	0.63992

Table 3. Average CCE at epoch 500. Highlighted is the best performing technique.

All three **crossover techniques** were applied to test its effectiveness. From Table 1, 2, and 3 I observed that Single- and Two-point technique catch up with each other quite well even though Single-point technique outperform the other two in the numbers. I believe Single- and Two-point technique are generally more suitable for our use case because the **weights has to work together from neurons and layers perspective**. These two techniques design-wise preserve this property by swapping a segment of the sequence together.

Since I observed that Sing- and Two -Point technique perform quite close to each other and Uniform technique is struggling to catch up, subsequent experiments only consider Single- and Two-Point technique as its crossover technique.

Different Population Size

Population Size	G1	G2	G3	G4	G5	Average Accuracy
50	0.9475	0.9476	0.9414	0.9537	0.9506	0.94814
100	0.9599	0.9692	0.9537	0.9661	0.9692	0.96359
200	0.9661	0.963	0.963	0.9599	0.9599	0.96234

Table 4. Average accuracy at epoch 500. Highlighted is the best performing population size.

Population Size	G1	G2	G3	G4	G5	Average MSE
50	0.1409	0.1479	0.1576	0.1243	0.1328	0.1407
100	0.0947	0.0928	0.1196	0.1014	0.0918	0.1000

200	0.0835	0.0788	0.0919	0.088	0.0718	0.08279
-----	--------	--------	--------	-------	--------	---------

Table 5. Average MSE at epoch 500. Highlighted is the best performing population size.

Population Size	G1	G2	G3	G4	G5	Average CCE
50	0.3013	0.2802	0.3015	0.2548	0.2785	0.2832
100	0.2015	0.1956	0.2318	0.2086	0.2055	0.2086
200	0.161	0.1524	0.187	0.1702	0.1379	0.1617

Table 6. Average CCE at epoch 500. Highlighted is the best performing population size.

From Table 4, 5, and 6 I observed that larger population size brought forth great improvement in the performance. Both 100 and 200 score a very close accuracy, but 200 shows a greater improvement in terms of MSE and CCE.

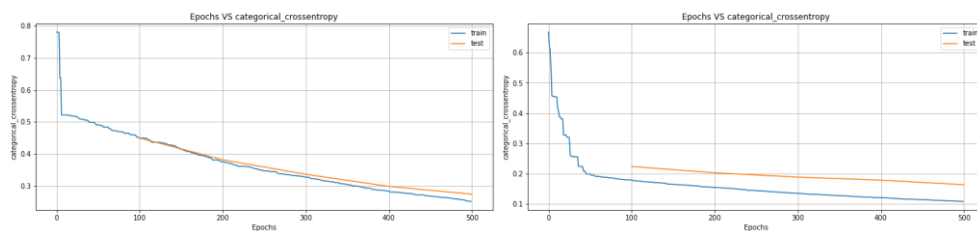


Figure 1. Learning curve (CCE) of population size 50 (left) and 200 (right) with single-point technique on sub dataset 5.

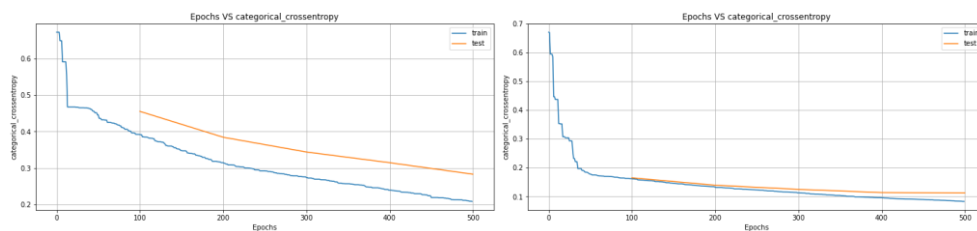


Figure 2. Learning curve (CCE) of population size 50 (left) and 200 (right) with two-point technique on sub dataset 5.

Hyperparameter Scaling

Scaled?	G1	G2	G3	G4	G5	Average Accuracy
False	0.9475	0.9476	0.9414	0.9537	0.9506	0.9481
True	0.9629	0.9722	0.9691	0.963	0.963	0.966

Table 7. Average accuracy at epoch 500. Highlighted is the best performing technique.

Scaled?	G1	G2	G3	G4	G5	Average MSE
False	0.1409	0.1479	0.1576	0.1243	0.1328	0.1407
True	0.0841	0.0689	0.0678	0.0784	0.0762	0.075

Table 8. Average MSE at epoch 500. Highlighted is the best performing technique.

Scaled?	G1	G2	G3	G4	G5	Average CCE
False	0.3013	0.2802	0.3015	0.2548	0.2785	0.2832
True	0.1485	0.1355	0.1318	0.1443	0.1417	0.1404

Table 9. Average CCE at epoch 500. Highlighted is the best performing technique.

I only propose one scaling policy which aim to reduce crossover rate, radioactive rating, and grow factor gradually along the training. I also make the mutation rate increase gradually along the training. Based on Table 7, 8 and 9 I observed the great benefits of hyperparameters scaling. I believe in the beginning of the training the optimizer needs sufficient **allowance to explore** the infinite search space, but towards the end the optimizer is already close to the global best therefore very **sensitive to adjustment**.

PSO

Vector Update Method

Mask?	G1	G2	G3	G4	G5	Average Accuracy
False	0.858	0.8704	0.8764	0.8395	0.8457	0.858
True	0.9691	0.9691	0.963	0.963	0.9691	0.9667

Table 10. Average accuracy at epoch 500. Highlighted is the best performing technique.

Mask?	G1	G2	G3	G4	G5	Average MSE
False	2.9058	2.9382	0.2422	2.9766	2.9529	2.4031
True	0.0583	0.0612	0.0748	0.0686	0.0679	0.0662

Table 11. Average MSE at epoch 500. Highlighted is the best performing technique.

Mask?	G1	G2	G3	G4	G5	Average CCE
False	0.5643	0.5686	0.4808	0.6425	0.6731	0.5859
True	0.1043	0.1146	0.1183	0.1085	0.1057	0.1103

Table 12. Average CCE at epoch 500. Highlighted is the best performing technique.

Based on the results shown in Table 11, 12, and 13 I observed the new method I proposed make significant improvement from the based version. From the train history, I observe that the optimizer converged around 200 epochs (see Figure 3).

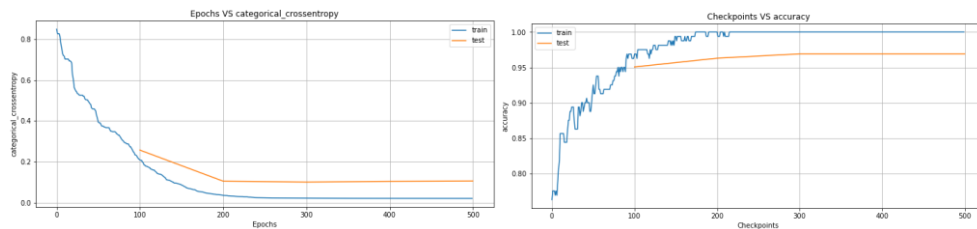


Figure 3. Loss function curve (left) and accuracy (right) with mask technique on sub dataset 5.

Shorter Training Time with Hyperparameter Scaling

Technique	G1	G2	G3	G4	G5	Average Accuracy
Epoch 500	0.9691	0.9691	0.963	0.963	0.9691	0.9667
E200 + Local	0.9753	0.9814	0.9568	0.9506	0.9691	0.9666
E200 + Global	0.963	0.9691	0.963	0.9444	0.963	0.9605

Table 13. Average accuracy. Highlighted is the best performing technique.

Technique	G1	G2	G3	G4	G5	Average MSE
Epoch 500	0.0583	0.0612	0.0748	0.0686	0.0679	0.0662
E200 + Local	0.0585	0.0484	0.0779	0.0898	0.0658	0.0681
E200 + Global	0.0695	0.0503	0.0769	0.1003	0.0655	0.0725

Table 14. Average MSE. Highlighted is the best performing technique.

Technique	G1	G2	G3	G4	G5	Average
-----------	----	----	----	----	----	---------

						CCE
Epoch 500	0.1043	0.1146	0.1183	0.1085	0.1057	0.1103
E200 + Local	0.1234	0.0851	0.1264	0.1452	0.1124	0.1185
E200 + Global	0.1259	0.1003	0.1209	0.1645	0.1156	0.1254

Table 15. Average CCE. Highlighted is the best performing technique.

Two type of hyperparameter scaling policies are tested here one that prioritize **local exploration** in the beginning and another one that **prioritize global chasing** in the beginning. Both are trained with only 200 epochs with learning rate also known as w scaled from 1.6 to 0.5 with *weight decay* 0.95 as the training progress. Hyperparameters were updated every 10 epochs. The one that prioritize local at beginning has $C1$ scaled from 2.0 to 0.5 and $C2$ scaled from 0.5 to 2.0 which the another one which prioritize global at the beginning has $C1$ scaled from 0.5 to 2.0 and $C2$ scaled from 2.0 to 0.5. It turned out scaling policy that prioritize local exploration in the beginning of training could keep up with model that trained for 500 epochs quite well with average accuracy of 0.9666 and average CCE of 0.1185.

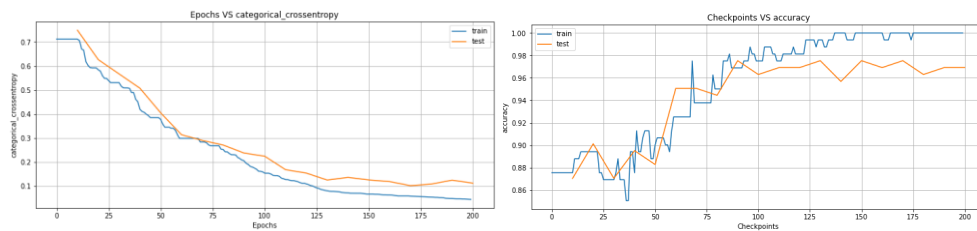


Figure 4. Loss function curve (left) and accuracy (right) of mask-PSO with 200 epochs and hyperparameters scaling on sub dataset 5.

Different Learning Rate

Learning Rate	G1	G2	G3	G4	G5	Average Accuracy
1.0	0.9691	0.9691	0.963	0.963	0.9691	0.9667
1.2	0.9753	0.963	0.9691	0.963	0.9691	0.9679
1.4	0.963	0.9753	0.963	0.9568	0.9815	0.9679

Table 16. Average accuracy. Highlighted is the best performing technique.

Learning Rate	G1	G2	G3	G4	G5	Average MSE

1.0	0.0583	0.0612	0.0748	0.0686	0.0679	0.0662
1.2	0.0628	0.0617	0.0645	0.0758	0.0713	0.0672
1.4	0.0736	0.0643	0.0659	0.0857	0.0871	0.0753

Table 17. Average MSE. Highlighted is the best performing technique.

Learning Rate	G1	G2	G3	G4	G5	Average CCE
1.0	0.1043	0.1146	0.1183	0.1085	0.1057	0.1103
1.2	0.1202	0.1091	0.105	0.125	0.1361	0.1191
1.4	0.1345	0.1173	0.1026	0.1414	0.1339	0.1259

Table 18. Average CCE. Highlighted is the best performing technique.

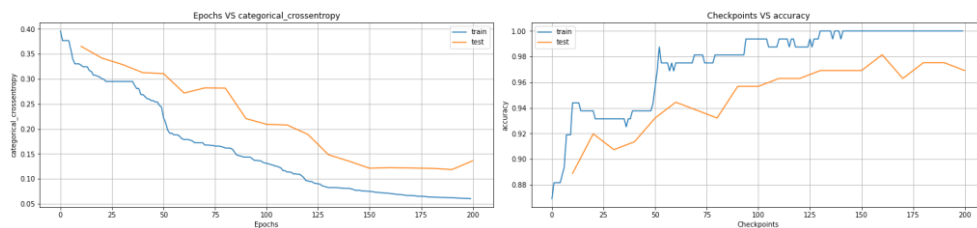


Figure 5. Loss function curve (left) and accuracy (right) of mask-PSO with 200 epochs and learning rate 1.2 on sub dataset 5.

I also tested different learning rates w 1.0, 1.2, and 1.4. All these models are trained for 200 epochs (except models with learning rate 1.0) and with constant $C1$ and $C2$ of 2.0 and population size of 50. Models trained with learning rate 1.2 and 1.4 were trained for 200 epochs. With just 0.2 difference in learning rate, I already observed performance changes. Learning rate 1.2 seems like a good balance between 1.0 (**careful player**) and 1.4 (**risk taker**). Although both 1.2 and 1.4 scored the same average accuracy Of 0.9679 but 1.2 also scored a much lower average CCE of 0.1191.

Different Population Size

As observed from previous experiments, scaled hyperparameter and constant hyperparameter both achieve reasonably well results. Different population will be tested on these two approaches.

Population Size	G1	G2	G3	G4	G5	Average Accuracy
50	0.9753	0.9814	0.9568	0.9506	0.9691	0.9667

100	0.9691	0.9691	0.9753	0.9691	0.963	0.9691
200	0.9815	0.9753	0.9568	0.9568	0.9691	0.9679

Table 19. Average accuracy. Highlighted is the best performing population size.

Population Size	G1	G2	G3	G4	G5	Average MSE
50	0.0585	0.0484	0.0779	0.0898	0.0658	0.0681
100	0.0651	0.0686	0.0905	0.07	0.0728	0.0734
200	0.0493	0.0643	0.0688	0.0939	0.067	0.0687

Table 20. Average MSE. Highlighted is the best performing population size.

Population Size	G1	G2	G3	G4	G5	Average CCE
50	0.1234	0.0851	0.1264	0.1452	0.1124	0.1185
100	0.1088	0.1246	0.1553	0.111	0.1247	0.1245
200	0.0955	0.1125	0.1094	0.152	0.1151	0.1169

Table 21. Average CCE. Highlighted is the best performing population size.

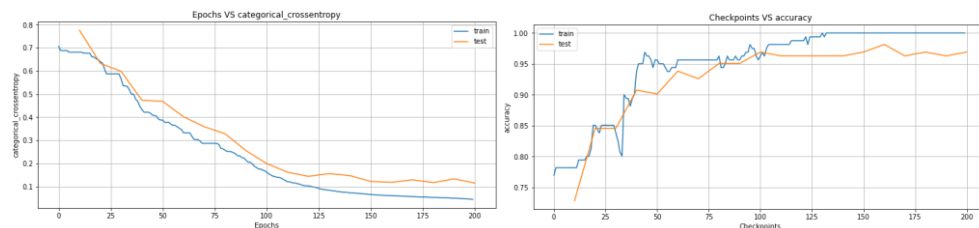


Figure 6. Loss function curve (left) and accuracy (right) of mask-PSO

with 200 epochs and population size 200 with hyperparameter scaling (Local) on sub dataset 5.

Models were trained for 200 epochs with the hyperparameters configuration and scaling policy stated before which prioritize local exploration at the beginning of training to test the difference between different population size. Although models with population size 100 score a higher average accuracy 0.9691, but it also has higher average MSE 0.0734 and average CCE 0.1245. The best performing population size should be 200 because its has **balanced performance** in terms of accuracy and loss.

Population Size	G1	G2	G3	G4	G5	Average Accuracy
50	0.9753	0.963	0.9691	0.963	0.9693	0.9679

100	0.963	0.963	0.9568	0.9568	0.9691	0.9617
200	0.963	0.963	0.9568	0.9691	0.9753	0.9654

Table 22. Average accuracy at epoch 500. Highlighted is the best performing population size.

Population Size	G1	G2	G3	G4	G5	Average MSE
50	0.0628	0.0617	0.0645	0.0758	0.0517	0.0633
100	0.0714	0.0739	0.0813	0.0794	0.071	0.0754
200	0.0622	0.0669	0.0861	0.0759	0.0639	0.071

Table 23. Average MSE at epoch 500. Highlighted is the best performing population size.

Population Size	G1	G2	G3	G4	G5	Average CCE
50	0.1202	0.1091	0.105	0.125	0.0803	0.1079
100	0.1231	0.1306	0.1252	0.1381	0.112	0.1245
200	0.1102	0.1194	0.1421	0.1253	0.1028	0.12

Table 24. Average CCE at epoch 500. Highlighted is the best performing population size.

I also tested different population size on models without hyperparameter scaling. These models were trained for 200 epochs with learning rate 1.2. The results surprised me because the models in this configuration **did not improved with a larger population size**. I could not explain this finding, the best guess I can make is larger population size causes more hopping around in the search space. Frequent hopping around in the later stage of training with large steps might cause the model to miss the global best solution. More experiments should be conducted to verify my guess.

Population size 50 without hyperparameters scaling remain the best performing configuration which score average accuracy of 0.9679 and average CCE of 0.1079. This configuration is much superior to the configuration with population size 200 because larger population size would increase the computational cost greatly.

Regularization

L1 or Lasso regularization is a regularization term added to the loss of the model with respect to its weights. It was often coupled with logistic regression for feature selection because it turns less influential features to zero weights. This behavior is particularly useful in feature reduction. Below is the mathematical expression of L1 regularization, λ is a constant that decides how much of regularization to be applied. It should be a reasonably small positive value so that it does not handicap the model's learning.

$$L1 = \lambda * \frac{1}{n} \sum_{i=0}^n |w_i|$$

L2 or Ridge regularization is also a regularization term added to the loss of the model with respect to its weights. It was often used to prevent overfitting of a machine learning model by keeping all weights small. The optimizer was wired to find the weights sequence with the least loss. Once the optimizer finds a sweet spot, it dives in as deep as it can. However, a model with extensively long training typically does not generalize well because it suits the training set too well including the noise like outliers. As a result, we can expect to see weights that are extremely large or extremely small from the optimizer. While having normalized inputs could prevent issues caused by great coefficient differences between features, having weights that are within a narrower range could also produce a much more stable model output. Below is the mathematical expression of L2 regularization, λ is a constant that decides how much of regularization to be applied. It should be a reasonably small positive value so that it does not handicap the model's learning.

$$L2 = \lambda * \frac{1}{n} \sum_{i=0}^n w_i^2$$

As expected, L1 aligned very well with models without a regularization term, and models with L2 performed poorer as compared to the other two. However, it's worth noting that all three methods have average accuracy above 90% in such a small dataset. I also inspected the weight representation of each model and included them in the experiment script attached. More weights that are close to the weight boundary we set are seen in models with L1 regularization and without regularization while models with L2 regularization generally have weights that are closer to zero.

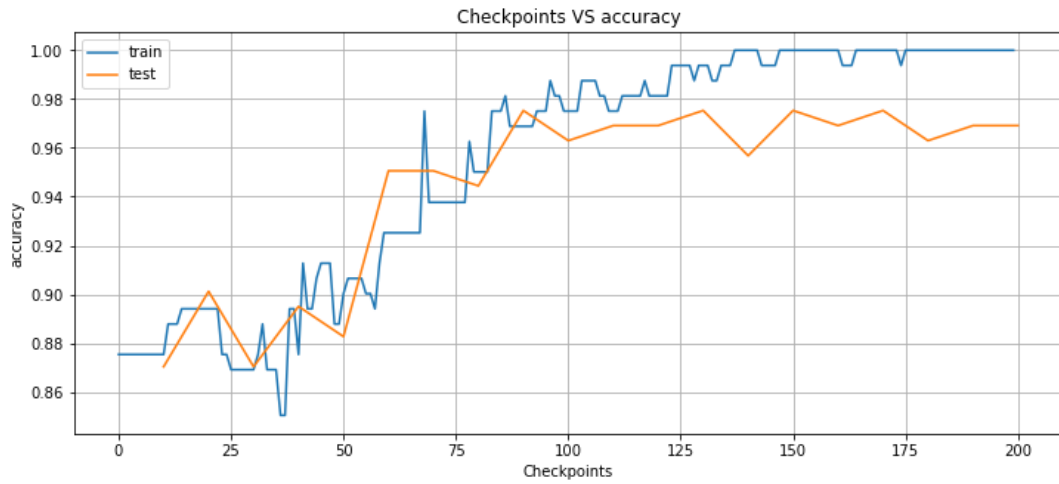


Figure 7. Accuracy record of scaled mask-PSO.

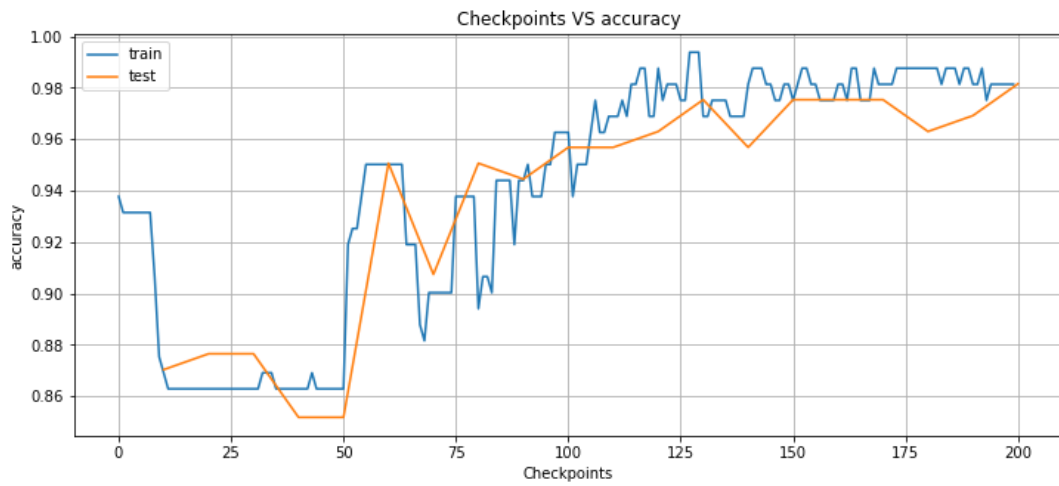


Figure 8. Accuracy record of scaled mask-PSO with l1-regularization.

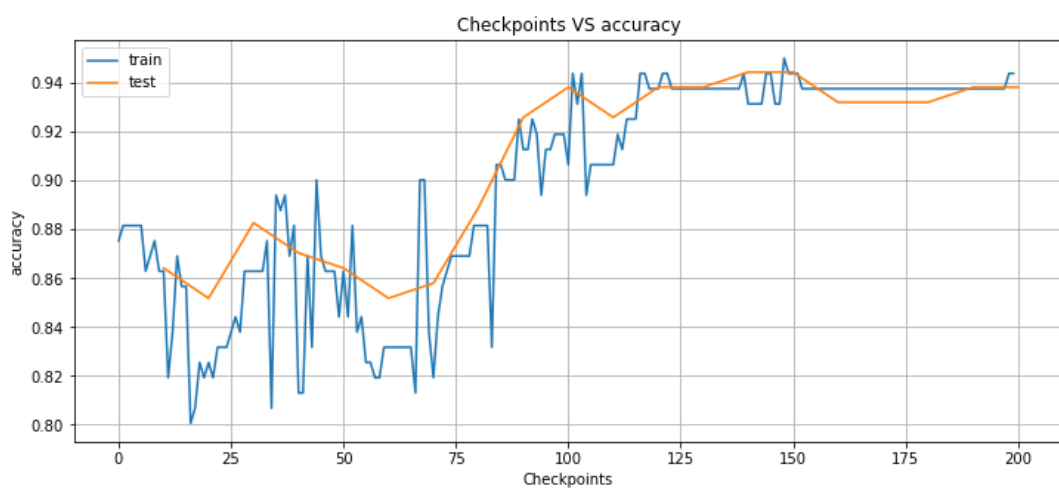


Figure 9. Accuracy record of scaled mask-PSO with l2-regularization.

*Figure 7, 8, and 9 were captured from training with sub dataset 5.

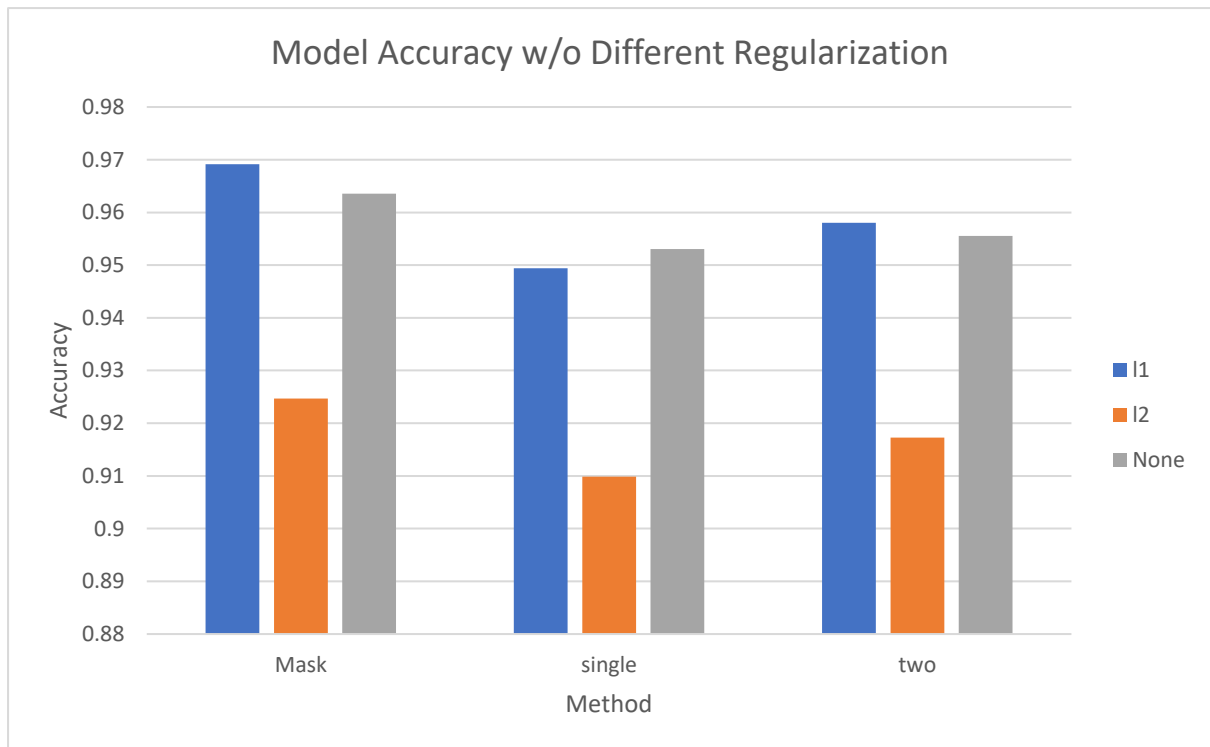


Figure 10. Comparison of average accuracy of different regularization on different models. Results were captured at epoch 200 and the configuration of these models has population size of 50 and uses hyperparameters scaling (Local for mask-PSO).

Conclusion

Putting the best configured version of both optimizers side by side, we could see that they actually catch up with each other quite well. However, mask-PSO is generally a better option for our case because it not only achieved highest average accuracy and lowest average CCE but also much efficient in terms of execution and intuitive in terms of implementation.