1. **The proposed algorithm (IP-PSO)**
2. Algorithm Overview

**Algorithm 1:** Framework of IP-PSO

*P* <- Initialize the population with the proposed particle encoding strategy

*t* <- 0

*P\_id* <- empty

*P\_gd* <- empty

**While** termination criterion is not satisﬁed **do**

**for each** *ind* **in** *P* **do**

*ind* <- update *ind* using the particle update equation

**if** termination criterion is satisfied **then**

**break**

**end**

**end**

**end**

**# Report P\_gd as the best solution**

Algorithm 1 outlines the framework of the proposed algorithm. There are mainly three steps which are really straightforward - initialise the population by using the particle encoding strategy which will be described in Part B, update the position and velocity and check whether the termination criterion meets.

1. Particle Encoding Strategy

IP-PSO encoding strategy is derived from the Network IP addresses. Since CNN is comprised of Convolutional Layer, Pooling Layer, and Fully-Connected Layer and the encoded information of different types of layers varies in terms of both the number of fields and the range in each field shown in Table 1-1 to Table 1-3, a fixed length of the Network IP with enough capacity can be designed to accommodate all the types of CNN layers and then the Network IP can be divided into numerous subsets each of which can be used to encode one type of CNN layers.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Layer Type | Field | Example | Range | # of Bits |
| Convolutional | Filter size | 2(2\*2) | 1-32 | 5 |
|  | # of feature maps | 32 | 1-128 | 7 |
|  | Stride size | 2(2\*2) | 1-16 | 4 |
|  | Mean value | 1.28 | (-255 to 256)/100 | 9 |
|  | Standard deviation | 2 | (1 to 512)/100 | 9 |
| Total |  |  |  | 34 |

**Table 1-1**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Layer Type | Field | Example | Range | # of Bits |
| Pooling | Kernel size | 2(2\*2) | 1-32 | 5 |
|  | Stride size | 2(2\*2) | 1-16 | 4 |
|  | Type | 1(average), 2(maximal) | 1-8 | 3 |
| Total |  |  |  | 12 |

**Table 1-2**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Layer Type | Field | Example | Range | # of Bits |
| Fully-Connected | # of Neurons | 32 | 1-512 | 9 |
|  | Mean value | 1.28 | (-255 to 256)/100 | 9 |
|  | Standard deviation | 2 | (1 to 512)/100 | 9 |
| Total |  |  |  | 27 |

**Table 1-3**:

First of all, the length of the IP liked encoding binary string needs to be designed. As the largest number of bits to represent a layer is 34 and the length of the others are much shorter, there is only one more bit that is required; However, in order to leave some capacity for further extensions, I will add 2 bits which brings up the total length of the encoding IP address to 36 bits.

In addition, the subnets for all types of layers need to be defined and CIDR(Classless Inter-Domain Routing) style will be used to represent the subnet. The subnet **0.0.0.0.0/2** with the range from 0.0.0.0 to 3.255.255.255.255 which has the capacity of 34 bits will be used encode the convolutional layer, the subnet **4.0.0.0.0/9** with the range from 4.0.0.0.0 to 4.7.255.255.255 which has the capacity of 27 bits will represent the fully-connected layer, and the subnet **4.8.0.0.0/24** with the range from 4.8.0.0.0 to 4.8.0.15.255 which has the capacity of 12 bits will carry the information of the Pooling layer.

Last but not least, as the particle length of PSO is fixed after initialisation, in order to cope with the variable length of CNN architecture, I will use an alternative way of disabling some of the variables in the encoding vector. Therefore another subnet **4.8.0.16.0/24** with the range from 4.8.0.16.0 to 4.8.0.31.255 will be introduced to mark the layer as not used.

To summarise, the subnet table where the subsets designed to represent the different types of CNN layers can be drawn as Table 2 and each layer will be encoded into a 5 bytes IP address.

|  |  |  |
| --- | --- | --- |
| Layer type | Subnet(CIDR) | IP Range |
| Convolutional Layer | 0.0.0.0.0/2 | 0.0.0.0 - 3.255.255.255.255 |
| Fully-Connected Layer | 4.0.0.0.0/9 | 4.0.0.0.0 - 4.7.255.255.255 |
| Pooling Layer | 4.8.0.0.0/24 | 4.8.0.0.0 - 4.8.0.15.255 |
| Disabled Layer | 4.8.0.16.0/24 | 4.8.0.16.0 - 4.8.0.31.255 |

**Table 2**:

After converting each layer into a 5 bytes IP address, the position and velocity of PSO can be designed. However, there are a few parameters that need to be defined first - max\_length(maximum length of CNN layers), max\_fully\_connected(maximum fully-connected layers given at least there is one fully-connected layer) listed in Table 3. And then the encoded data type of the position and the velocity will be a byte array with a fixed length of maximum\_length \* 5 and each byte will be deemed as one dimension of the particle.

|  |  |
| --- | --- |
| Parameter Name | Parameter Meaning |
| max\_length | maximum length of CNN layers |
| max\_fully\_connected | maximum fully-connected layers given at least there is one fully-connected layer |
| N | population size |
| k | the training epoch number before evaluating the trained CNN |
| num\_of\_batch | the batch size for evaluating the CNN |
| c1 | acceleration coefficient array for *P\_id* |
| c2 | acceleration coefficient array for *P\_id* |

**Table 3**: Parameter list

1. Population Initialisation

**Algorithm 2**: Population Initialisation

**Input:** the population size N, the maximum length of CNN layers max\_length, the maximum fully-connected layers max\_fully\_connected

**Output:** Initialised population *P\_0*

*P\_0* <- initialise an empty byte array

**While** | *P\_0* | < N **do**

*particle* <- Initialise a byte array

***# add conv, pooling or disabled layer into the particle***

**While** | *particle* | < 5 \* (*max\_length* - *max\_fully\_connected*) **do**

*r* <- Uniformly generate a number between [0, 1]

**If** *r* < 0.34 **then**

*L* = Initialise a random IP address in the subnet of convolutional layer

**else if** *r* < 0.68 **then**

*L* = Initialise a random IP address in the subnet of pooling layer

**else**

*L* = Initialise a random IP address in the subnet of disabled layer

**end**

*particle* <- append *L* at the end of *particle* byte array

**end**

*Is\_fully\_connected* <- **false**

***# start add fully-connected layer by chance when the length left is not greater than***

***# max\_fully\_connected***

**While** | *particle* | > 5 \* (max\_length - max\_fully\_connected) **and**

| *particle* | < 5 \* (max\_length - 1) **do**

**If** *Is\_fully\_connected* == **true**

*L* = Initialise a random IP address in the subnet of fully-connected layer

**else**

*r* <- Uniformly generate a number between [0, 1]

**If** *r* < 0.25 **then**

*L* = Initialise a random IP address in the subnet of convolutional layer

**else if** *r* < 0.50 **then**

*L* = Initialise a random IP address in the subnet of pooling layer

**else if** r < 0.75 **then**

*L* = Initialise a random IP address in the subnet of disabled layer

**else**

*L* = Initialise a random IP address in the subnet of fully-connected layer

*Is\_fully\_connected* <- **true**

**end**

*particle* <- append *L* at the end of *particle* byte array

**end**

***# always have a fully-connected layer at the end***

*L* = Initialise a random IP address in the subnet of fully-connected

*particle* <- append *L* at the end of *particle* byte array

*P\_0* <- append *particle* at the end of *P\_0*

**end**

In terms of the population initialisation, we set up the size of the population and randomly create individuals until reaching the population size.

For each individual, first we initialise the vector with the length of max\_length \* 5 since 5 bytes are the unit to represent a layer, and then split the vector into four parts by three boundaries - 1, (max\_length - max\_fully\_connected) \* 5 and 5 \* (max\_length - 1). The first unit will always be a convolutional layer, the second part can be filled with convolutional layer, pooling layer or disabled layer, the third part can be filled with any of the four types of layers, and the last part which is the last unit layer will always be a fully-connected layer. In addition, each layer will be generated with the random settings.

1. Fitness Evaluation

**Algorithm 3**: Fitness Evaluation

**Input:** The population *P\_t*, the training epoch number *k* for measuring the accuracy tendency, the training set *D\_train*, the ﬁtness evaluation dataset *D\_fitness*, and the batch size num\_of\_batch

**Output**: The population with ﬁtness P\_t

**for each** individual *s* **in** *P\_t* **do**

*i* <- 1

*eval\_steps* <- | *D\_fitness* | / *num\_of\_batch*

**while** *i* <= *k* **do**

Train the connection weights of the CNN represented by individual *s*

**if** *i* == *k* **then**

*accy\_list* <- empty array

*j* <- 1

**while** *j* <= *eval\_steps* **do**

*accy\_j* <- Evaluate the classiﬁcation error on the j-th batch data from *D\_fitness*

*accy\_list* <- append *accy\_j* at the end of *accy\_list*

*j* <- *j* + 1

**end**

Calculate the number of parameters in *s*, the mean value and standard derivation from *accy\_list*, assign them to individual *s*, and update *s* from *P\_t*

**end**

i <- i + 1

**end**

**end**

**Return** *P\_t*

With regard to the fitness evaluation, each individual is decoded to a CNN with its settings which will be trained for k epoch on the training dataset, and then the trained CNN will be batch-evaluated on the validation dataset which will produce a set of accuracies. Finally, we calculate the mean and standard deviation of the accuracies for each individual which will be stored as the individual fitness along with the number of parameters of the CNN.

For comparing the fitness of individuals, the mean value, standard deviation and the number of parameters will be used in order for the comparison, ie. compare mean value first, if mean value is equal compare standard deviation, if standard deviation is the same compare the number of parameters.

1. Particle Update Equation

**Algorithm 4**: Update Equation

**Input:** particle individual *ind*, acceleration coefficient array for *P\_id* *c1*, acceleration coefficient array for *P\_id* *c2*

**Output**: updated individual *ind*

**for each** *d* => (*x, v*) **in** *ind* **do**

*c\_index* <- *d* % 5

*r1* <- uniformly generate *r1* between [0, 1]

*r2* <- uniformly generate *r2* between [0, 1]

*v\_new* <- *v* + *c1*[*c\_index*] \* *r1* \* (*P\_id* - *x*) + *c2*[*c\_index*] \* *r2* \* (*P\_gd* - *x*)

*x\_new* <- *x* + *v\_new*

**if** x\_new falls **in** allowed subnets for current dimension **then**

x\_new <- replace *x\_new* with a random IP in the next available subnet

**end**

*ind[d]* <- update *x, v* to *x\_new, v\_new*

**end**

*P\_current* <- evaluate the updated individual *ind*

**if** P\_current > P\_id **then**

P\_id <- P\_current

**end**

**if** P\_current > P\_gd **then**

P\_gd <- P\_current

**end**

**Return** *ind*

As each layer is encoded into a unit with 5 dimensions in the particle vector and we want to control the acceleration coefficients for each dimension, the two acceleration coefficients are two float arrays with the length of 5. In order to leverage local search - in other words not to change the layer type too often, the smaller value will be assigned to the upper dimensions on the left side of the IP representation and a typical set of coefficients can be [0.00001, 0.0001, 0.001, 0.01, 0.1].

After the coefficients defined, we go through each dimension in the individual and update the velocity and position by using the corresponding coefficients for that dimension. Since there are some constraints for each part of the four parts of the vector, e.g. the second part can only be convolutional layer, pooling layer or disabled layer, we need to upgrade the new position by a random IP in the next available subnet if the position does not fall in a valid subnet. And then we evaluate the new position, compare it with its local best and the global best, and update the two bests if needed.

1. Best Individual Selection and Decoding

Global best of PSO will be reported as the best individual. In terms of the decoding, we can first identify the type of the layer represented by the IP address - stored in every five bytes/dimensions from left to right in the position vector of the global best, according to the subnets in Table 2, and then based on Table 1-1 to Table 1-3 we can decode the IP into different sets of bits which indicate different fields of the layer. After decoding all the IP addresses in the global best, the final CNN architecture can be obtained by connecting all of the decoded layers in the same order as the IP addresses in the position vector.

1. placeholder
2. **Placeholder**