**Objective**

Use Hadoop Map-Reduce to evolve Genetic Algorithm by parallelizing the operations and output the solution for the target given the various parameters like chromosome length, mutation rate, crossover rate and population size.

**Assumption**

In genetic algorithm, the two good (but not best) chromosomes should be selected so that the good characteristics of parents are included in Crossover and best will remain in the population. For this implementation, we are assuming that the chromosomes which have the high fittest score will crossover and mutate to give the best output solution.

**Solution**

Based on the serial GA, we can naturally come up with the Mapper and Reducer design. Several GA parallelization strategies exist depending on the grain of parallelization to achieve. Basically, three levels of parallelization can be exploited:

* fitness evaluation level (i.e., global parallelization model);
* population level (i.e., coarse-grained parallelization or island model);
* individual level (i.e., fine-grained parallelization or grid model).

In the global parallelization model, a node acting as a master, manages the population (i.e., applying genetic and selection operators) and distributes the individuals among slave nodes which compute only the fitness values of the individuals. The main advantage of using such a model is that it does not require any change to the design of traditional GA since the individual fitness evaluation is independent from the rest of the population.

The underlying idea in using MapReduce to parallelize the Genetic Algorithm is to encapsulate each iteration of the GA as a separate MapReduce job and parallelize the chromosome fitness evaluation assigning such task to several Mappers, while a single Reducer is responsible to collect the results and to perform the genetic operations (i.e., parent’s selection, crossover and mutation, and survival selection)

needed to produce a new generation following a global parallelization model. The MapReduceJob is the

core of the Parallel Genetic Algorithm module since it allows us to parallelize fitness evaluations and

distribute the computation over the nodes. A MapReduceJob consists of three phases (i.e., Split, Map, and

Reduce) in which each component performs its proper task.

In the split phase, the InputFormat module gets the current population and processes it in order to split it in crunch of data (i.e., input split) to be distributed among the Mapper modules.

In the map phase, each Mapper carries out its task on the received input split in a parallel and independent way.

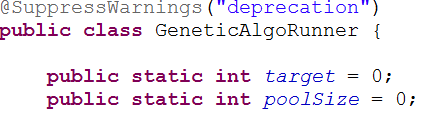
Once such evaluation is completed, each Mapper generates a new pair, where value is a pair, while the new key will be used by the Master module to properly assign the Reducers.**Code Explanation**

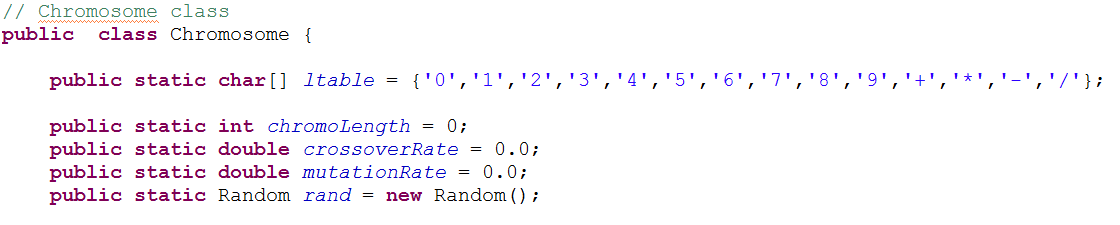
1. **Read the Input File**

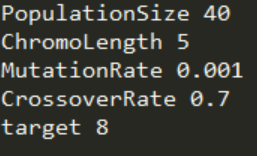
In the first MapReduce job, we have only mapper, the **ReadMapper** is going to read the input file which will have parameters like population size, crossover rate, mutation rate, chromosome length and target. We are parametrizing or taking the inputs from the users, which will make the job flexible. Mutation rate (**mutationRate**) should be very low, we have took it as **0.001** for running the algorithm. Crossover rate (**crossoverRate**) of **0.7** is good for the Genetic Algorithm. Population size we have taken as **40**, but we have run it for various population size, it is called as **poolSize**. Chromosome length which we are considering for this iteration is **5** and we are calling it as **chromoLength**. Chromosome length should be odd, because then only expression will be valid. For expression should be valid the number should be followed by an operator and end in a number. We have a function called **isValid()**  for validating the function in the Chromosome class. We are also defining the **target** value in the input file, what target we want’s to achieve using genetic algorithm, the value we have taken is **8**.

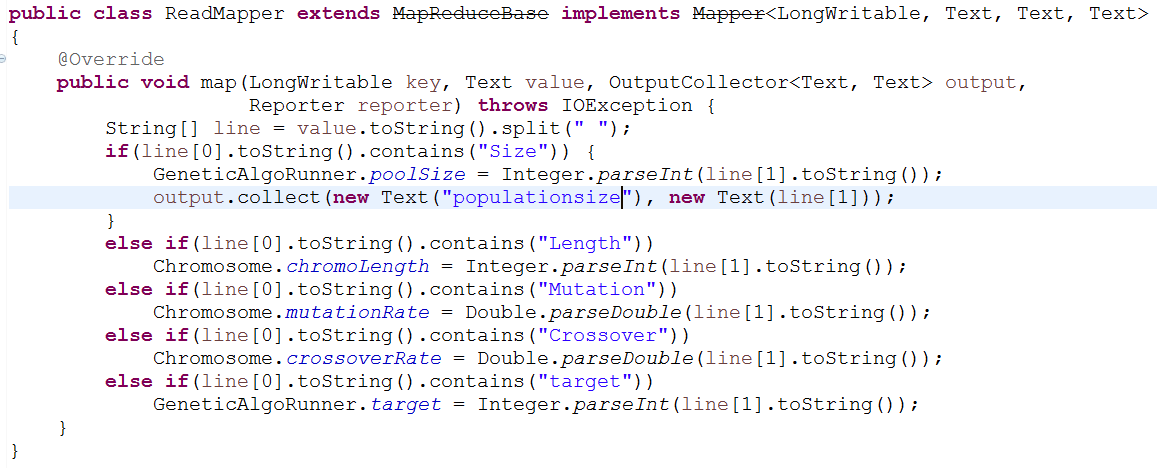


We are going to read the input file and set the value of global variable which we have defined in Chromosome and GeneticAlgoRunner class. Mutation Rate, crossover rate, chromosome length are declared in Chromosome class, Pool size and target in GeneticAlgoRunner class





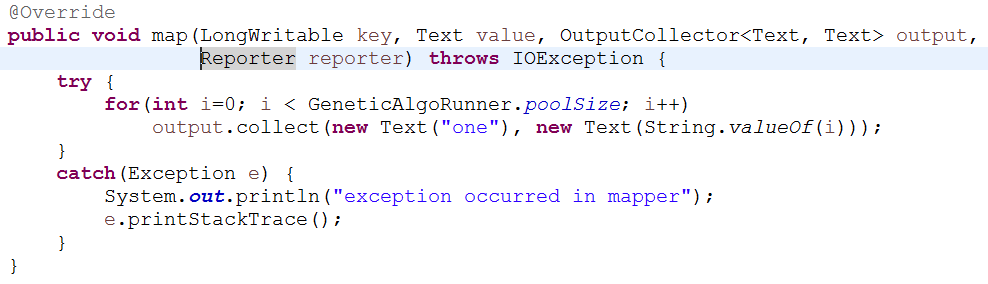




1. **Creation of unique chromosomes**

In the second MapReduce Job, in mapper, we are going to create new chromosome and compute the fitness score given the population size as 40 and each chromosome of 4 bit.

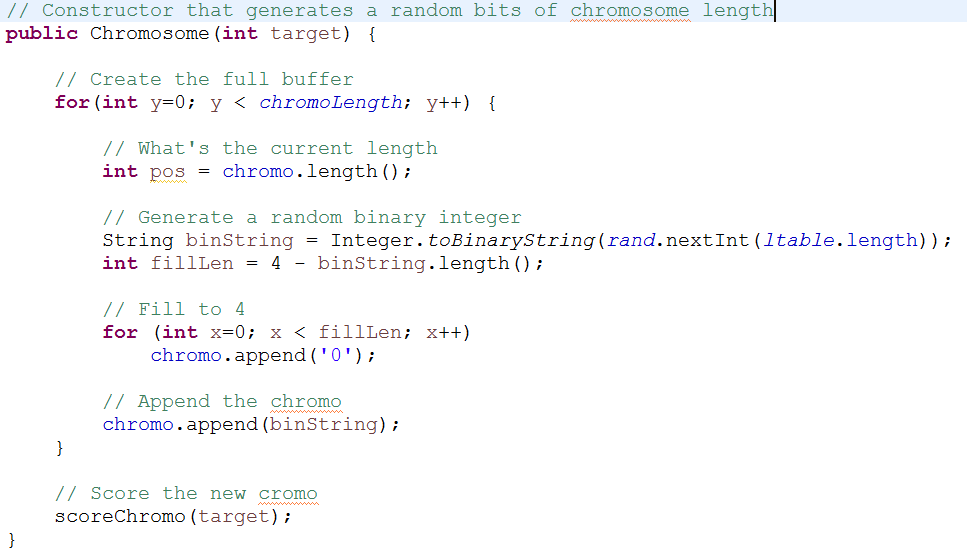
In **ChromosomeMapper**, we are going to iterate through the size of the population and emit the “**one**” as key and the chromosome number i.e, “**index number”** as value. We are emitting all the population with the same key because the population is not that big and we can generate all the chromosome in a single place.There is no use taking them in multiple reducers since our population size is not that huge. If in future, we want to take the larger population size we can employ multiple reducers so that we can distribute our work in that case.



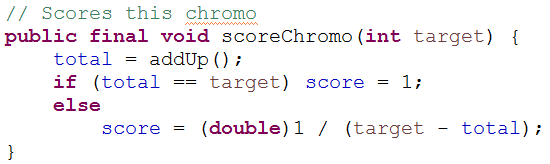
**Input**: population size (refer to pool size in code)

**Output**: Chromosome numbers to be given to one reducer

In **ChromosomeReducer**, we are going to generate new Chromosomes by looping over the chromosome number which were emitted by the mapper. After generating new chromosomes, we can compute the score of the chromosome. The below function is going to generate a random bit and create a chromosome of chromosome length defined based on the target value.



Once the chromosome is generated, we have a function called **scoreChromo()** which is going to compute the fitness score of the chromosome. The general idea is to give a higher fitness score the closer a chromosome comes to solving a problem. The score of the chromosome is **1 / (target – total)** where target is the expected value and total is computed value of that chromosome. The closed we get to a solution, the closed we get to a 1. Therefore, when target == total, then the score will be one. Assume the target number of the remainder of the story is 42 and the chromosome is 23, then the score of that chromosome will be **1 / (42 – 23) = (1 / 19)**.

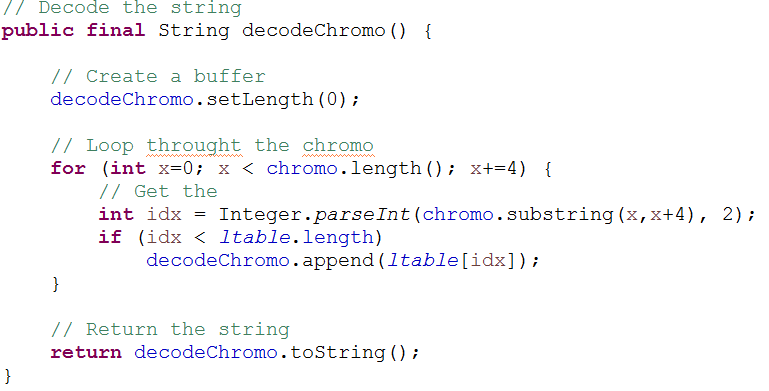


We are going to use **addUp()** function, to compute the total of the chromosome, so that we can compute the fitness score. The **addUp() function** is going to use the **decode()** function, in which we are going to use **ltable**. Ltable is a character array having value **{'0','1','2','3','4','5','6','7','8','9','+','\*','-','/'}.** We are going to take the 4 bits of the chromosome and convert it into decimal and then decode it to one of the element from the ltable. In our use case **1 is represented 0001, 2 as 0010, 9 as 1001, ‘+’ as 1010, ‘\*‘ as 1011, ‘-‘ as 1100 and ‘/’ as 1101**.



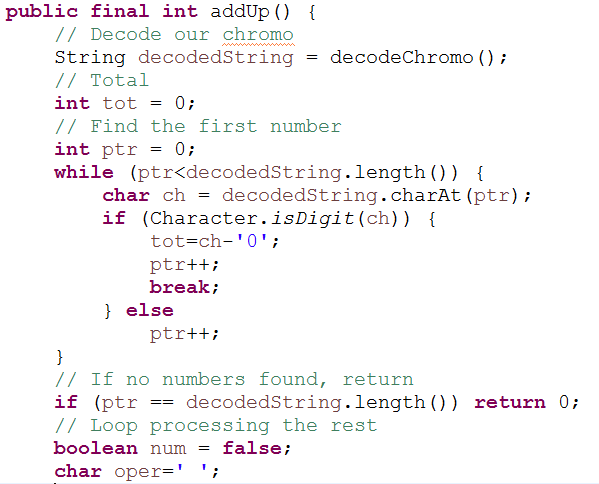
**Decode Function**: Decode the chromosome from binary string to number and operator.

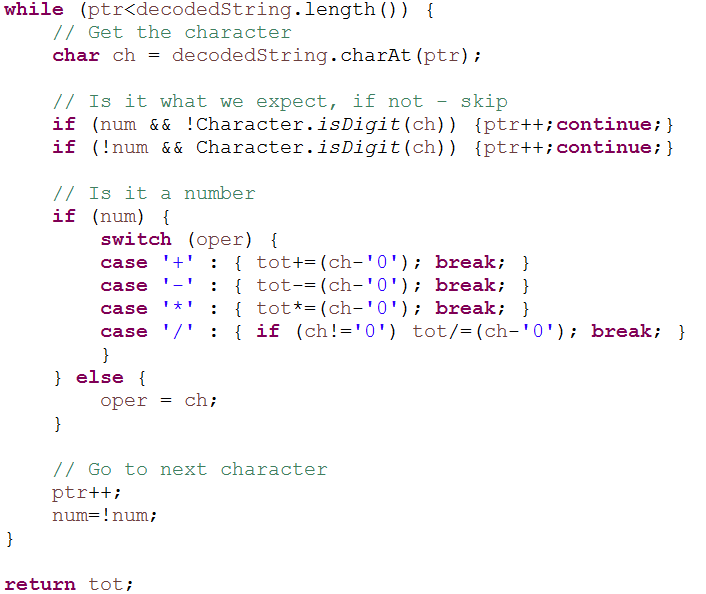
**Example**: **0110 1010 0110 1100 0100 1101 0010 1010 0001. Decoded as: 6 + 6 – 4 / 2 + 1**



**AddUp Function**: is going to add the decode chromosome and operator will be applied sequentially from left to right.

The addUp function will generate the **value as 5** for the above chromosome.





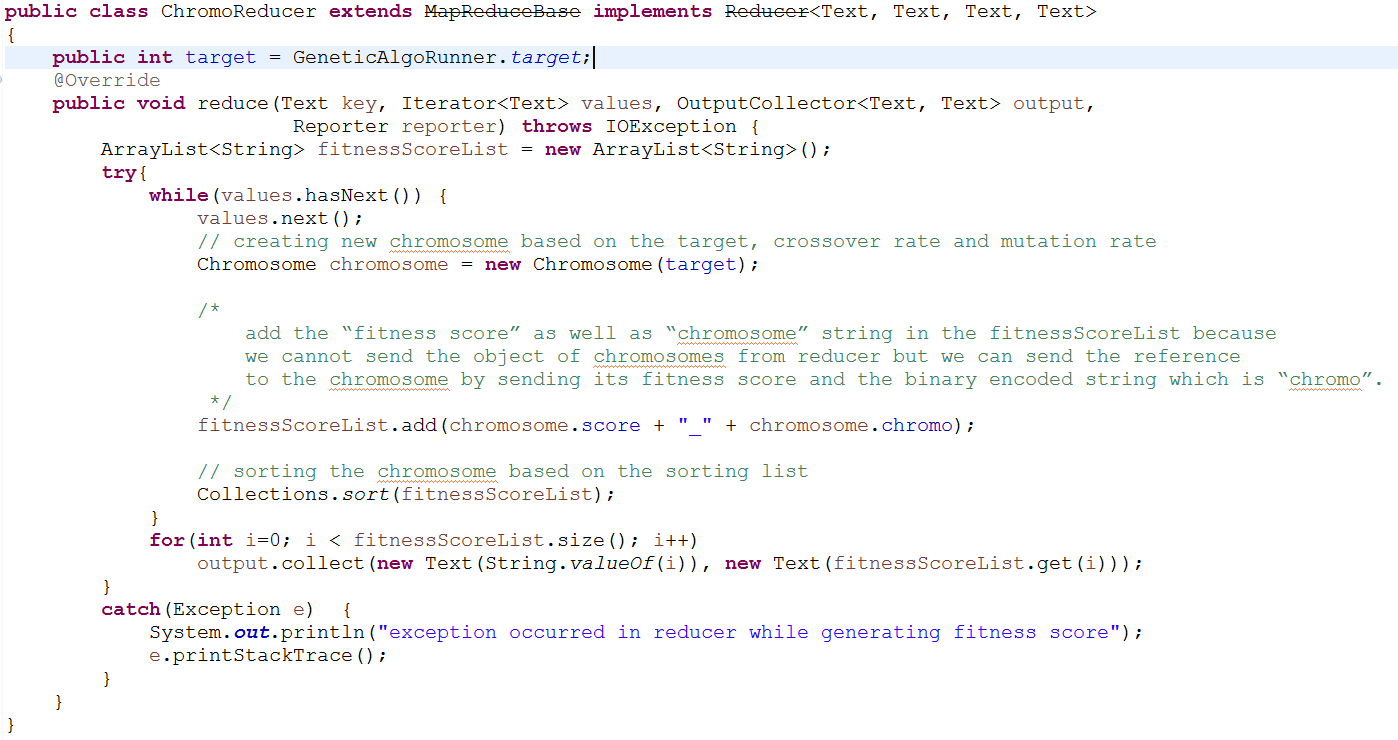
Once we have the chromosome and the fitness score, we will create the fitness score list which will have chromosome number that is basically a line number or **“index of population” as key** and **values as “chromosome score\_chromosomeBits”**. We have to add the “fitness score” as well as “chromosome” string in the fitnessScoreList because we cannot send the object of chromosomes from reducer but we can send the reference the chromosome by sending its fitness score and the binary encoded string which is “chromo”. After putting this, we are going to sort the list based on the fitness score. The keys will be 0 to 39 for the iteration since the we have population size of 40.

**Input**: Chromosome numbers which work just like line numbers

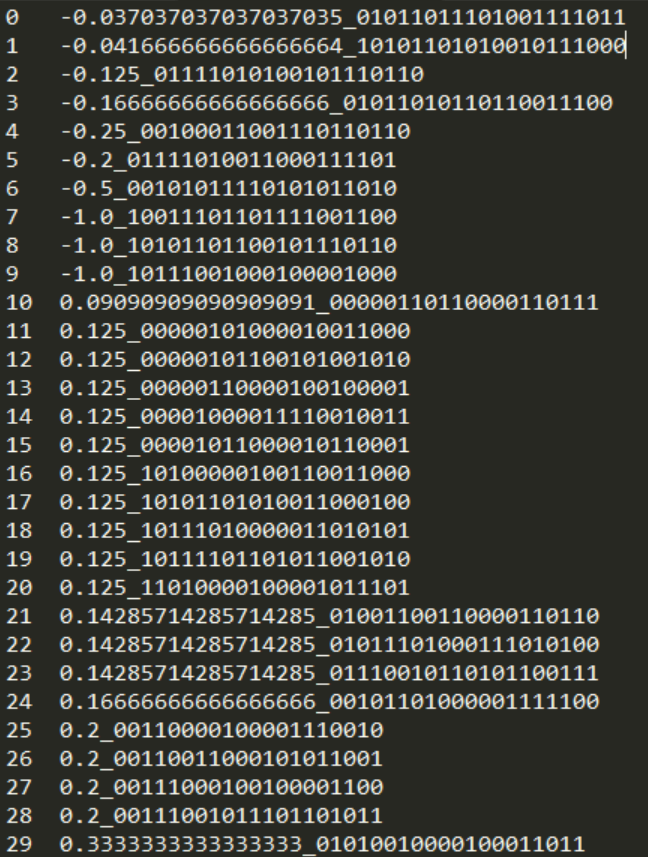
**Output**:

**Key:** chromosome number

**Value:** fitnessScore\_chromo, Example: (0.16666666666666666\_00100010001001010111)



**Screenshot of the Output (Output Directory)**

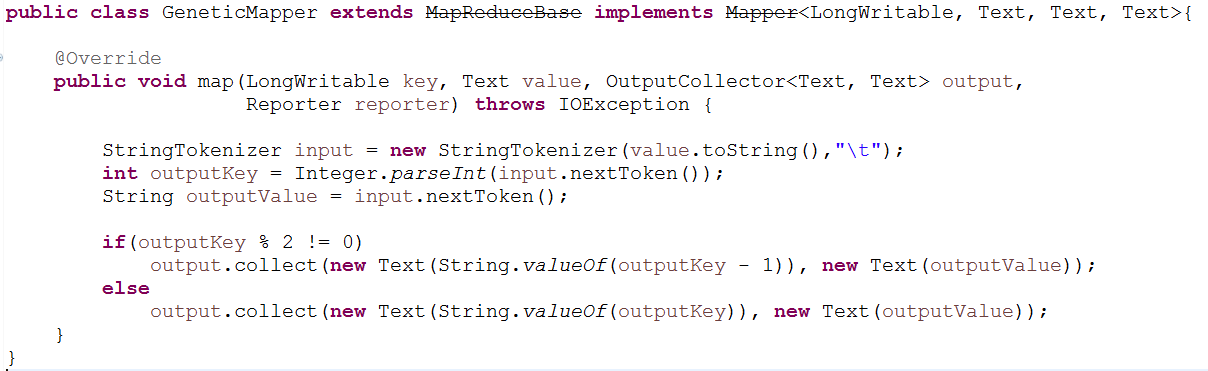


1. **Send the most probable chromosomes to reproduce and mutate to the same Reducer**

* Select two members from the current population, chances of chromosome being selected is proportional to the chromosome fitness score
* Choose bits from each chosen chromosome at a randomly chosen, dependent on the crossover rate, crossover the bits
* Crossover: is performed by selecting a random gene along the length of the chromosomes and swapping all the genes after that point
  + Crossover rate: the chance that two chromosomes will swap their bits. CrossoverRate is 0.7
* Mutation: Step through the chosen chromosomes bits and flip dependent on the mutation rate
  + Mutation rate: this is the chance that a bit within a chromosome will be flipped (0 becomes 1, 1 becomes 0). This is ususally a very low value for binary encoded genes, say 0.001

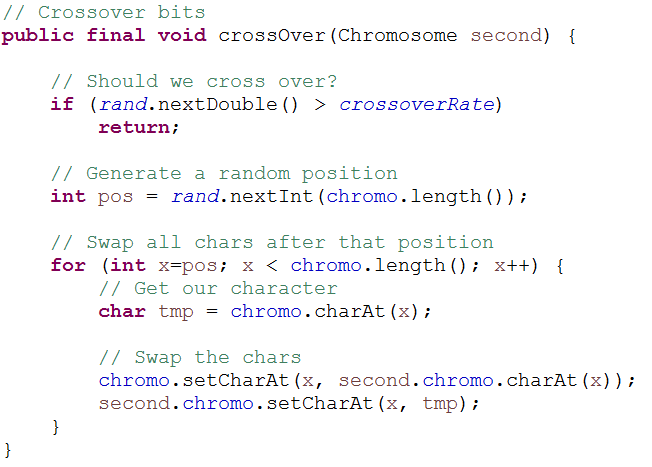
In **GeneticMapper**, we are going to receive the input which was emitted as output from the Chromosome Reducer, it will be in form line number as key and “fitnessScore\_chromo” as value. Since the input will be sorted as per the fitness score, we will emit the same key for consecutive chromosomes. We do this because we want to have the fittest chromosomes to mutate in order to give best output. The chromosomes which are in even place (line number) will emit the **key** that they have been associated

within the output of the above MapReduce. The chromosomes which are in odd place (line number) will emit the **(key – 1)** as key so that they go to the same reducer as their consecutive counterpart. **The number of reducers required will be half of the population size (**because we are sending to pairs to each reducer**).**

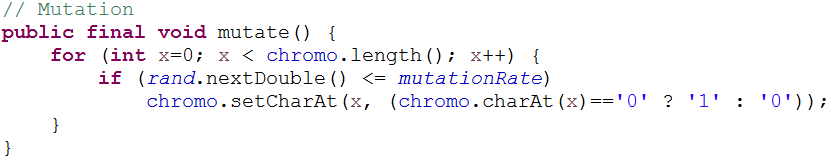


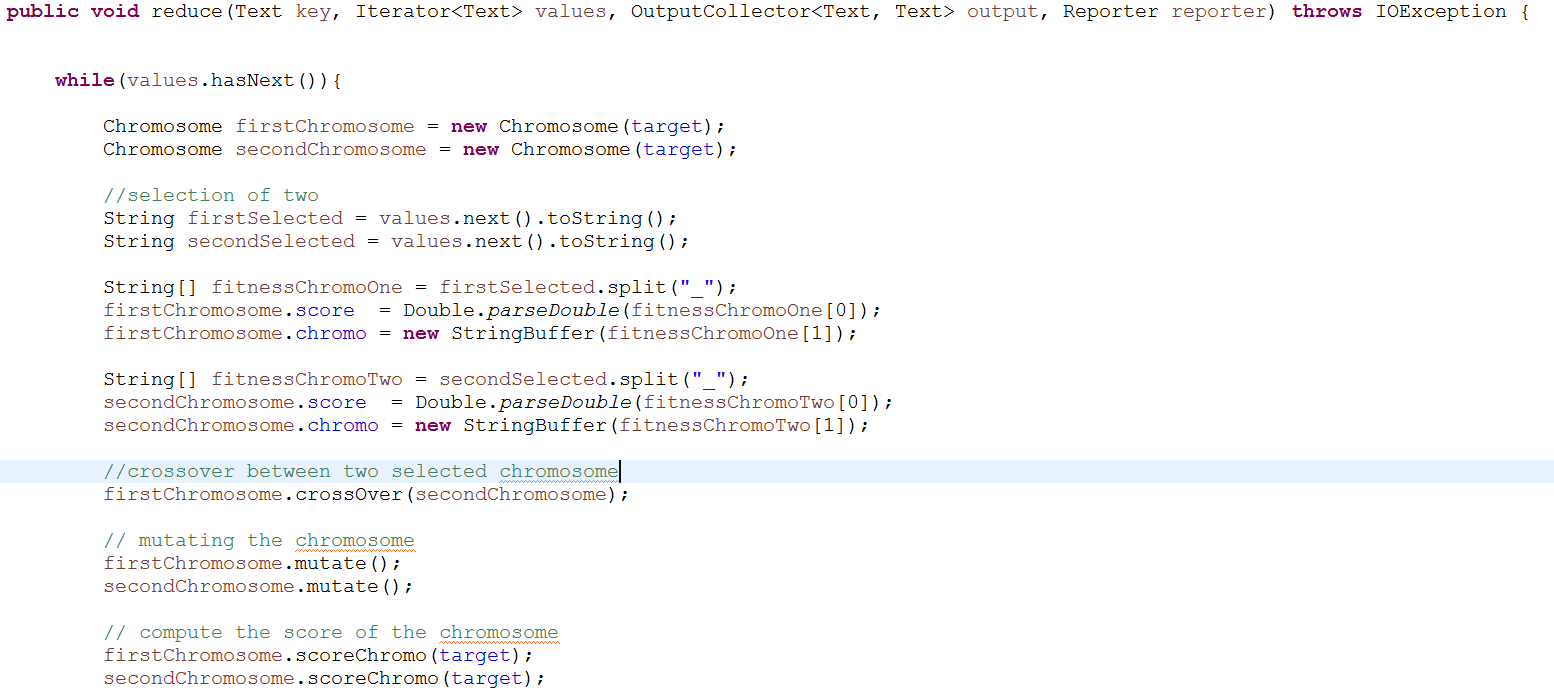
**In GeneticReducer,** we are going to get the pair of chromosomes (Fitness score and chromo which is binary encoded string) as input. After receiving the chromosome, we are going to create chromosome object and the fitness score and chromo to two different chromosome. After doing that, the next step is to perform genetic algorithm

* Crossover the two chromosomes, based on crossover rate which is read from the input file in the ReadMapper



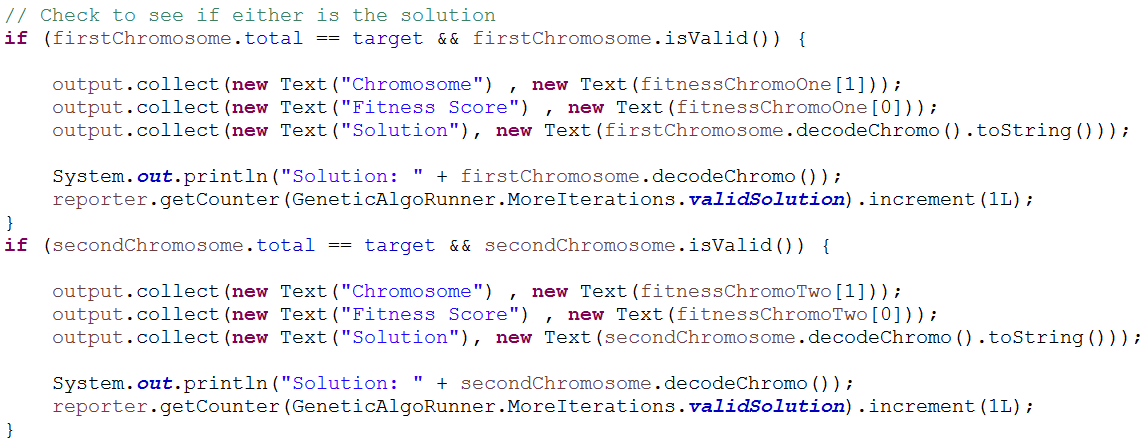
* Mutate the two chromosomes, based on mutation rate which is read from the input file in the ReadMapper





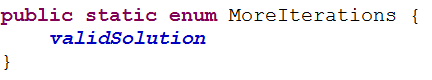
* Check if the total score is equal to the target and if it is valid chromosome (based on number -> operator -> number -> operator -> number)
* Iterate till we get the best chromosome mutation

Once we get the best chromosome, we are going to decode the binary string and output the **“Solution” as key** and **“Decoded chromosome” as value.** Along with that we are going to output the Fitness score and the chromosome.



**Iteration**

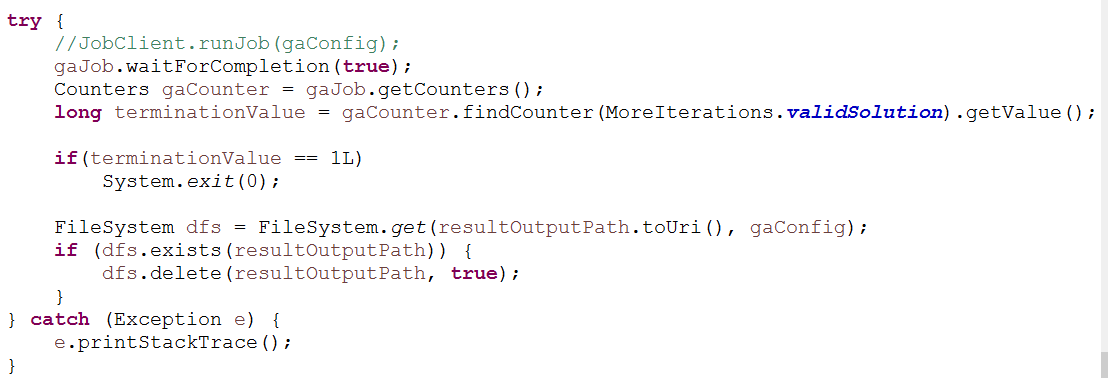
The genetic algorithm is iterative algorithm until we found a good solution, we are going to run the Chromosome MapReduce and Genetic MapReduce in a loop, until we found an appropriate solution. For stopping this Iterative MapReduce, we are using counters, counters are global variable which are defined as enum and are accessed globally in Mapper, Reducer and Driver/Runner.



Once we found an appropriate solution, we are going to increment its value in **GeneticReducer**



And we are constantly check for its value in **GeneticAlgoRunner** class, as soon as we found its value as 1. We are going to stop the iteration once that job is completed.

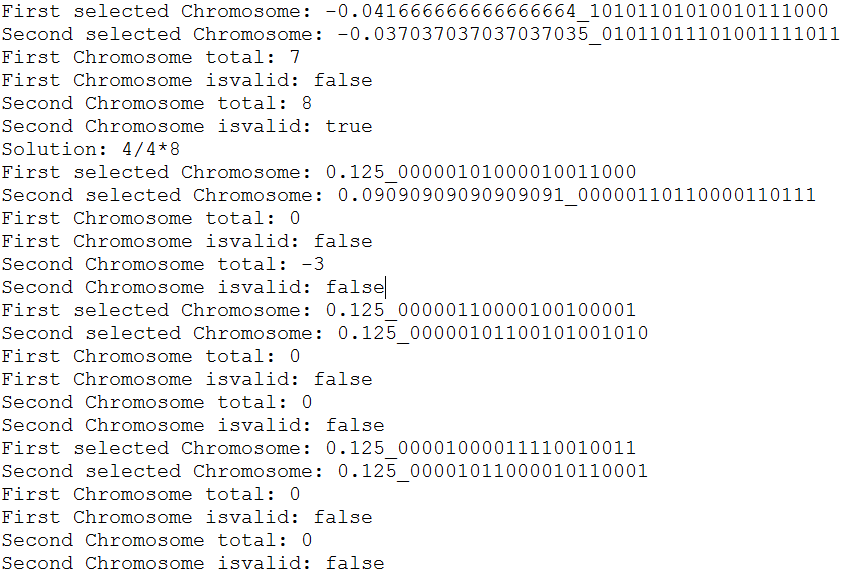


**Conclusion**

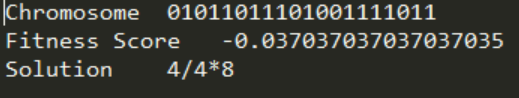
Genetic Algorithm is very difficult to run on Hadoop because it involves a lot of computations and takes a lot of iteration to find a single solution. Hence we do not prefer running genetic algorithm on Hadoop.

**Screenshots**

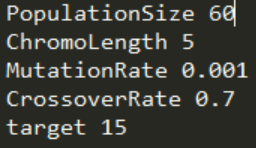
**Console Output**



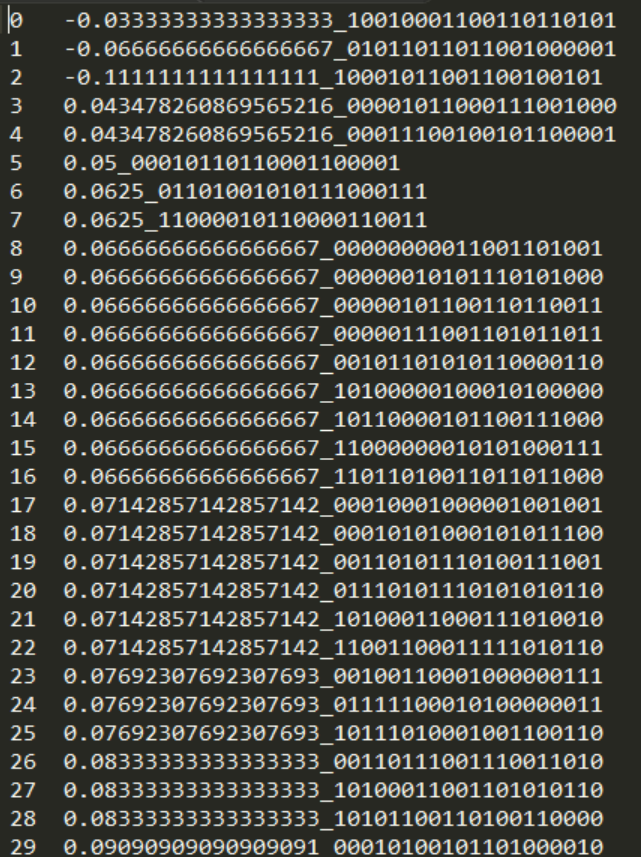
**Final Output (Result Directory)**



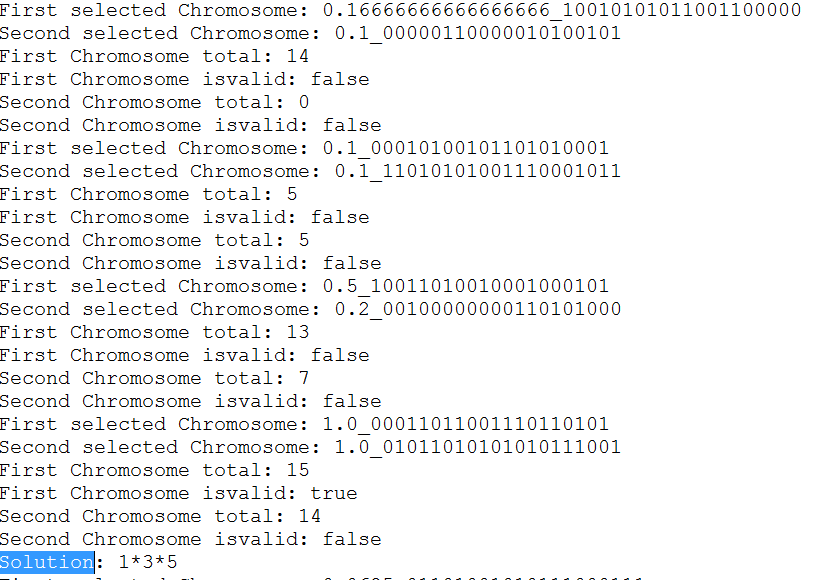
**Test Case 2**



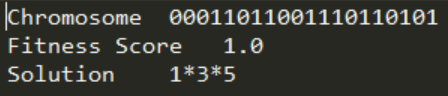
**Chromosome MapReduce Output (Output Directory)**



**Console Output**



**Genetic MapReduce Output (Result Directory)**



**Ques. How could you accomplish a similar task using an Artifical Neural Network (ANN)**

**Solution:**

In the Artificial Neural Network, the architecture of a neural network and the learning rate of the algorithm used to train the model are important decisions when seeking to solve a task. We can take the help of Genetic Algorithm to evolve the topology of neural networks.

However, this methodology has **two main challenges:**

1. **Competing Convention Problem**

There is a possibility that two genomes represent the same solution but they have different encoding; i.e., two neural networks that order their hidden nodes differently in their genomes might still be functionally equivalent. This is a problem because differing representations of the same structure are highly likely to produce a damaged offspring during crossover.

**Example:** consider two functionally equivalent genomes with different structures, represented as [P, Q, R] and [R, Q, P]. A useful method of performing crossover during a genetic algorithm is to use single-point crossover, in which a randomly chosen recombination point is used to split each genome into two parts, one of which is swapped with the other parents. Using single-point crossover with our two selected genomes yields [P, Q, P] and [R, Q, R]. Unfortunately, both [P, Q, P] and [R, Q, R] are missing a hidden node that was (presumably) important to the solution. Thus, both offspring are likely not good candidate solutions that the algorithm will now needlessly evaluate. Therefore, crossing over two equivalent genomes will produce the damaged children.

1. **Topological Innovation**

There are the fitness implications from adding new or adjusting existing topology of the network. Often, such a change will initially decrease fitness before the connection weight is given an opportunity to optimize. If that genome was culled from the population before it was given an opportunity to reproduce in order for its children to develop the new structure further, we might be inadvertently halting a promising structural development. The solution for the topological innovation is ***speciation***, a process to ensure that genomes will compute within a population niche. Two organisms are said to be in the same species if they represent topologically similar networks.

**Algorithm**

The algorithm which we are going to address is the NEAT Algorithm (NeuroEvolution of the Augmenting Topologies) seeks to resolve the challenges, including evolving neural network topologies along with weights. NEAT proves to be effective due to:

1. Employing a principled method of crossover of different topologies (competing convention problem)
2. Protecting structural innovation using speciation (topological innovation)
3. Incrementally growing from minimal structure

The very first step in evolving ANN using genetic algorithm is to encode the network architecture in the GA chromosome. **Encoding Scheme,** Genomes are represented in a list-like data structure that contains ***node genes*** and ***connection genes***. **Connection genes** contain information about the nodes it connects, the weight of the connection, whether the connection is enabled, and a ***historical marker*** that provides information about the ancestral history of the gene. Genes that share a historical origin necessarily represent the same structure in a neural network phenotype. Thus, each new gene that is created through structural mutation is assigned a unique ***global innovation number***. When individual genes are copied to an offspring genome, they retain their historical marker. This allows genomes to be compared by matching up genes that share a historical marker (and thus represent the same structural component in possibly differing larger network structures). Thus, historical marking seeks to answer the previous questions regarding viable offspring and topological innovation by framing them as topological matching problems.

1. **Crossover**

During crossover, genes in each genome are aligned based on their historical marker in a process called ***artificial synapsis***. Genes that share an ancestral origin are called ***matching genes***; those that don’t are called ***disjoint genes or excess genes*** depending on if they exist inside or outside the range of innovation numbers of the other parent, respectively. Thus, although two parents might look different, their ***historical markers allow us to find structural similarities without any (likely expensive) topological analysis***. Excess genes and disjoint genes are inherited from the more fit parent.

***Artificial synapsis*** ***helps counteract the negative side effects caused by the competing conventions problem***. Compatible organisms mate in a way which allows for the preservation of functional subunits; offspring are likely to be undamaged. Although competing conventions might still exist in the population, NEAT does not spend time evaluating damaged offspring, and thus the main consequence of competing conventions is avoided.

1. **Speciation**

Genomes are allowed to compete with populations niches by grouping them into species. Although this appears to be a topological matching problem, historical markers again provide a useful solution. The topological discrepancy between two neural networks can be quantified with the number of excess and disjoint genes between their genomes. ***The more disjoint the genomes, the less evolutionary history they share, and thus the less compatible they are***.

1. **Mutation**

**Mutation rate** is the important parameter in the NEAT algorithm because they determine how finely or coarsely the algorithm searches over the search space. Large topological changes are equivalent to large movements over the fitness landscape; if these changes are excessively large, the algorithm will not be able to converge on an optimal or near-optimal solution. There are four main types of mutation operations in NEAT:

* Add a neuron
* Add a connection
* Remove a node
* Remove a connection

**When adding a neuron**, a random connection is selected and replaced by a new neuron and two new connections. The weights of the two new connections are selected to be effectively the same as the replaced connection in order to avoid overly-drastic movements over the fitness landscape.

**Adding a new connection** works in a similar way; two neurons are selected at random, between which a new connection is added. Checks are performed to ensure a connection doesn’t already exist between the source and destination neuron. This also highlights two interesting features of NEAT: ***hidden neurons aren’t ordered in a traditional hierarchy of hidden levels and recurrent neural networks are allowed to evolve (and ideally will, if that will solve the problem more efficiently than a feed-forward neural network)***. When adding anew connection, the source and destination neuron chosen are allowed to bethe same neuron.

***The removal of neurons and connections is performed similarly. Both connections and neurons can be randomly selected for removal***. Hidden neuronsare removed if there are no remaining connections interacting with them. When a ***synaptic weight*** is randomly chosen to be perturbed, a random number from a zero-centered normal distribution is added to the weight.

1. **Fitness Function**

After each epoch, a measure of the discrepancy between the observed and predicted values of the dependent variable is calculated. Often, this discrepancy is expressed as a ***mean square error*** (MSE), which for this study was the error function:

**E(g) = (1 / N) \* [summation of (ˆyi − yi) \* (ˆyi − yi) from i=1 to N]**

where E(g) is the mean square error term, N is the number of observation (input) vectors presented to the network, ˆyi is the predicted response, and yi is the observed response. The N observations constitute the training data set. ***Fitness is simply 1−E(g); a perfect organism would have a fitness equal to 1***. Theoretically, NEAT should find the genome that represents the neural network that minimizes E(g) (i.e. maximizes fitness).

**Ques. How could you leverage mapreduce to train your ANN**

**Solution:**

In ANN, Input signals first come to first layer, as known as the input layer, and directly transfer to the middle/hidden layer through weighted connections. The incoming signals are operated by each specific neuron in the hidden layers; in such way output values transfer to each neuron in the output layer through a second layer of weights. At last, via the calculation and operation of the output layer, the output signal is produced. Apparently, the hidden layer may include several layers. The learning procedures aim at adjusting the weights in the artificial neural network model, so that the performance of models is advanced over time. A many-layered neural network could be effectively pre-trained one layer at a time, treating each layer in turn as un unsupervised Restricted Boltzmann Machine (A restricted Boltzmann machine (RBM) is a generative stochastic artificial neural network that can learn a probability distribution over its set of inputs.), then followed by supervised back-propagation fine-tuning.

To leverage the Map-Reduce, we should read the data and extract the key/values pairs and transmit them as a list of intermediate key/value pairs which is latter used by reducer to compute the output. The input value will be the data from a certain source, that a machine is going to “learn”, and in our case, **the random chromosomes are created in a form of binary string based on the population size**. The intermediate key/value pair will be the weights, in order for a machine to determine whether it has acknowledged the object correctly. Reducer function uses these weights to compute the so-called “acknowledgment” of the machine of an intended object (**target value** in our case). If there exits an intolerable difference between the precision of training set and expected precision, the MapReduce job will loop until an acceptable result has worked out. This is basically an Iterative MapReduce, we can compute the acceptability of the solution based on the mean square error (also known as **fitness score**), once we found the solution, we can emit that value and stop the iteration.

The **main challenge** we are going to face with this approach is the large amount of data weighting to be trained and deep learning is not capable of eliminating the noises and similarity from the datasets. This may lead to loop the MapReduce many times and even after that it may not reach the satisfying precision and worst case, it may not even find the solution at all.

The **solution** to the above problem can be to remove the similar items by using diversity-based data sampling method. The **good solution** will be to somehow train the data so that the machine would find a certain pattern corresponded with the data set. Both are applicable to develop the efficiency of back-propagation.

**Example:** Suppose we want to find the chromosomes that representing a solution, instead of repeating chromosomes the good way is to find the chromosome and then decode the chromosome and find a solution. But to identify the chromosome we should first a way that help us identify the bits of the chromosome or identify the chromosome that belong to same species from the population, the techniques which we learned to overcome the challenges of NEAT algorithm.

**Algorithm**

1. **Pre-training**

The pre-training step is performed before fine-tuning is due to the unacceptable inefficiency of back propagation when converting a high-dimensional data into low-dimensional codes. When we have multiple layers, large initial weights always witnesses poor local minima, while small initial weights will lead to tiny gradients in early layers, making it infeasible to train the whole back propagation system with many hidden layers. Therefore, we have to make the initial weights close to the solution. We make advantage of restricted Boltzmann machine (RBM) to realize pre-training, where an ensemble of binary vectors can be modelled using a two-layer network, in which stochastic, binary pixels are connected to stochastic, binary feature detectors using symmetrically weighted connections. Pre-training involves learning a stack of RBMs, each of which has one layer of feature detectors, and it is sort of recursive process where the learned feature activations of one RBM are used as “data” for training the next RBM in the stack. One more point to be noticed is that we are adjusting the weights of paths based on the average variations of a single weight resulting from the modification made by a batch of training items. The reason is updating the value of weights once a single training item is extremely slow and inapplicable for MapReduce method.

1. **Fine tuning**

This is the step where we get zero and one from a circle and a stick. Pre-training initializes the weights so that they are close to the solution we want, but they are not the answer. We train the weights using back propagation algorithm based on MapReduce method in multiple layer neural network. Note that here the trained data is not randomized weights but well-initialized weights, making it easier to get the solution. Every mapper receives one training item and then computes all update value of the weights. Then each reducer gathers update-values for one weight and calculating the average. The input value of mapper is the input item while the input key is empty. On the other hand, the value of the reducer is the difference between original weight and updated weight (calculating mean squared error), and the corresponding key is the weight. Similar to pre-training, variations of each weight should be updated batch by batch for the sake of efficiency.

**MapReduce based Genetic Algorithm using ANN**

In our case, we are going to generate the random chromosome based on the population size given to us. Then we are going to divide the population by the number of mappers provided and we are going to distribute the batch of population to each mapper. Once each mapper will get the pool size, we are going to implement pre-training process handle the ***Competing Convention Problem***(There is a possibility that two genomes represent the same solution but they have different encoding).

The neural network will have one parameter namely ***epochs,*** which is basically the number of iteration times the neural network should be run. One iteration through the process of providing the network with a sample input and updating the network’s weights. The other parameters for training the ANN are **input feature size**, **number of hidden** and **output nodes**.

For genetic algorithm, the crossover rate, mutation rate and number of generation set at globally in the **Chromosome class**, which will be helpful during the creation of object. respectively. Furthermore, accuracy in classification is used to evaluate the fitness of populations. The **target** is declared in the Driver.

**In the Mapper**, propagate its set of training data through the network, and propagate errors to calculate the partial gradient for weight and as well going to invoke the **fitness function computation**, **adopting five-fold cross validation.** Three parameters of the ANN classifier (number of hidden nodes, learning rate and momentum), together with random number seeds, which might affect the initial weights and bias of neural network, **are encoded into the chromosomes**.

**In the Reducer**, sums the partial gradients and does a batch gradient descent to update the weights. In other words we are going to compute the sum of the neurons and then compare the sum with the target. Neurons are created by the chromosomes bit.

The **iterative MapReduce** is going to be in play until will we find a solution or in this case since we have define the epoch, that will be our exist condition. We can also increase the value of the epoch and implement **counters**, in MapReduce to keep the counter or end the job once we find the solution.

**References**

<http://ieeexplore.ieee.org/document/6200189/>

<http://digitalcommons.colby.edu/cgi/viewcontent.cgi?article=1836&context=honorstheses>

<https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=5&ved=0ahUKEwi4xKrl0abQAhVD54MKHdowAgkQFgg4MAQ&url=https%3A%2F%2Fwww.thinkmind.org%2Fdownload.php%3Farticleid%3Dadvcomp_2013_2_30_20081&usg=AFQjCNE7hyKDK9JB-ASwbxa2lEOUyUt-tg&sig2=LMEZht3XB4RN395Bn_kRMA&bvm=bv.138169073,d.amc&cad=rja>

<https://arxiv.org/pdf/1510.02709.pdf>