```cpp
#include "GA.h"

namespace SoilMath
{
    GA::GA() { }

    GA::GA(NNfunctionType nnfunction, uint32_t inputneurons, uint32_t hiddenneurons, uint32_t outputneurons)
    {
        this->NNfuction = nnfunction;
        this->inputneurons = inputneurons;
        this->hiddenneurons = hiddenneurons;
        this->outputneurons = outputneurons;
    }


    GA::~GA() { }

    void GA::Evolve(const ComplexVect_t &inputValues, Weight_t &weights, std::vector<Weight_t> &prevWeights, MinMaxWeight_t
      rangeweights, Predict_t goal, uint32_t maxGenerations, uint32_t popSize)
    {
        // Create the population
        uint32_t NOprevPopUsed = prevWeights.size() < popSize ? prevWeights.size() : popSize;
        Population_t pop = Genesis(weights, rangeweights, popSize - NOprevPopUsed);
        for (uint32_t i = 0; i < NOprevPopUsed; i++)
        {
            PopMember_t newMember;
            newMember.weights = prevWeights[i];
            for (uint32_t j = 0; j < newMember.weights.size(); j++) { newMember.weightsGen.push_back(ConvertToGenome<float>
              (newMember.weights[j], rangeweights)); }
            pop.push_back(newMember);
        }
        float totalFitness = 0.0;
        for (uint32_t i = 0; i < maxGenerations; i++)
        {
            CrossOver(pop);
            Mutate(pop);
            totalFitness = 0.0;
            GrowToAdulthood(pop, inputValues, rangeweights, goal, totalFitness);
            if (SurvivalOfTheFittest(pop, totalFitness)) { break; }
        }

        weights = pop[0].weights;
```

```
41      }
42
43
44      void GA::Evolve(const InputLearnVector_t &inputValues, Weight_t &weights, MinMaxWeight_t rangeweights, OutputLearnVector_t &goal,
            uint32_t maxGenerations, uint32_t popSize)
45      {
46          // Create the population
47          Population_t pop = Genesis(weights, rangeweights, popSize);
48          float totalFitness = 0.0;
49          for (uint32_t i = 0; i < maxGenerations; i++)
50          {
51              CrossOver(pop);
52              Mutate(pop);
53              totalFitness = 0.0;
54              GrowToAdulthood(pop, inputValues, rangeweights, goal, totalFitness);
55              if (SurvivalOfTheFittest(pop, totalFitness)) { break; }
56          }
57          weights = pop[0].weights;
58      }
59
60
61
62      Population_t GA::Genesis(const Weight_t &weights, MinMaxWeight_t rangeweights, uint32_t popSize)
63      {
64          if (popSize < 1) return Population_t();
65
66          Population_t pop;
67          unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
68          std::default_random_engine gen(seed);
69          std::uniform_real_distribution<float> dis(rangeweights.first, rangeweights.second);
70
71          for (uint32_t i = 0; i < popSize; i++)
72          {
73              PopMember_t I;
74              for (uint32_t j = 0; j < weights.size(); j++)
75              {
76                  I.weights.push_back(dis(gen));
77                  I.weightsGen.push_back(ConvertToGenome<float>(I.weights[j], rangeweights));
78              }
79              pop.push_back(I);
80          }
81          return pop;
```

```
82      }
83
84      void GA::CrossOver(Population_t &pop)
85      {
86          Population_t newPop; // create a new population
87          PopMember_t newPopMembers[2];
88          SplitGenome_t Split[2];
89
90          for (uint32_t i = 0; i < pop.size(); i += 2)
91          {
92
93              for (uint32_t j = 0; j < pop[i].weights.size(); j++)
94              {
95                  // Split A
96                  Split[0].first = bitset<CROSSOVER>(pop[i].weightsGen[j].to_string().substr(0, CROSSOVER));
97                  Split[0].second = bitset<GENE_MAX - CROSSOVER>(pop[i].weightsGen[j].to_string().substr(CROSSOVER, GENE_MAX -
                        CROSSOVER));
98
99                  // Split B
100                 Split[1].first = bitset<CROSSOVER>(pop[i + 1].weightsGen[j].to_string().substr(0, CROSSOVER));
101                 Split[1].second = bitset<GENE_MAX - CROSSOVER>(pop[i + 1].weightsGen[j].to_string().substr(CROSSOVER, GENE_MAX -
                        CROSSOVER));
102
103                 // Mate A and B to AB and BA
104                 newPopMembers[0].weightsGen.push_back(Genome_t(Split[0].first.to_string() + Split[1].second.to_string()));
105                 newPopMembers[1].weightsGen.push_back(Genome_t(Split[1].first.to_string() + Split[0].second.to_string()));
106             }
107             newPop.push_back(newPopMembers[0]);
108             newPop.push_back(newPopMembers[1]);
109             newPopMembers[0].weightsGen.clear();
110             newPopMembers[1].weightsGen.clear();
111         }
112
113         //Allow the top tiers population partners to mate again
114         uint32_t halfN = pop.size() / 2;
115         for (uint32_t i = 0; i < halfN; i++)
116         {
117             for (uint32_t j = 0; j < pop[i].weights.size(); j++)
118             {
119                 Split[0].first = bitset<CROSSOVER>(pop[i].weightsGen[j].to_string().substr(0, CROSSOVER));
120                 Split[0].second = bitset<GENE_MAX - CROSSOVER>(pop[i].weightsGen[j].to_string().substr(CROSSOVER, GENE_MAX -
                        CROSSOVER));
```

```
121
122                 Split[1].first = bitset<CROSSOVER>(pop[i + 2].weightsGen[j].to_string().substr(0, CROSSOVER));
123                 Split[1].second = bitset<GENE_MAX - CROSSOVER>(pop[i + 2].weightsGen[j].to_string().substr(CROSSOVER, GENE_MAX -
                      CROSSOVER));
124
125                 newPopMembers[0].weightsGen.push_back(Genome_t(Split[0].first.to_string() + Split[1].second.to_string()));
126                 newPopMembers[1].weightsGen.push_back(Genome_t(Split[1].first.to_string() + Split[0].second.to_string()));
127             }
128             newPop.push_back(newPopMembers[0]);
129             newPop.push_back(newPopMembers[1]);
130             newPopMembers[0].weightsGen.clear();
131             newPopMembers[1].weightsGen.clear();
132         }
133         pop = newPop;
134     }
135
136     void GA::Mutate(Population_t &pop)
137     {
138         unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
139
140         std::default_random_engine gen(seed);
141         std::uniform_real_distribution<float> dis(0, 1);
142
143         std::default_random_engine genGen(seed);
144         std::uniform_int_distribution<int> disGen(0, (GENE_MAX - 1));
145
146         for (uint32_t i = 0; i < pop.size(); i++)
147         {
148             for (uint32_t j = 0; j < pop[i].weightsGen.size(); j++) { if (dis(gen) < MUTATIONRATE) { pop[i].weightsGen[j][disGen
                  (genGen)].flip(); } }
149         }
150     }
151
152     void GA::GrowToAdulthood(Population_t &pop, const ComplexVect_t &inputValues, MinMaxWeight_t rangeweights, Predict_t goal, float
          &totalFitness)
153     {
154         for (uint32_t i = 0; i < pop.size(); i++)
155         {
156             for (uint32_t j = 0; j < pop[i].weightsGen.size(); j++) { pop[i].weights.push_back(ConvertToValue<float>(pop
                  [i].weightsGen[j], rangeweights));  }
157             Weight_t iWeight(pop[i].weights.begin(), pop[i].weights.begin() + ((inputneurons + 1) * hiddenneurons));
158             Weight_t hWeight(pop[i].weights.begin() + ((inputneurons + 1) * hiddenneurons), pop[i].weights.end());
```

```
159             Predict_t results = NNfuction(inputValues, iWeight, hWeight, inputneurons, hiddenneurons, outputneurons);
160             for (uint32_t j = 0; j < results.OutputNeurons.size(); j++)
161             {
162                 pop[i].Fitness -= results.OutputNeurons[j] / goal.OutputNeurons[j];
163             }
164             pop[i].Fitness += results.OutputNeurons.size();
165             totalFitness += pop[i].Fitness;
166         }
167     }
168
169     void GA::GrowToAdulthood(Population_t &pop, const InputLearnVector_t &inputValues, MinMaxWeight_t rangeweights,
          OutputLearnVector_t &goal, float &totalFitness)
170     {
171         for (uint32_t i = 0; i < pop.size(); i++)
172         {
173             for (uint32_t j = 0; j < pop[i].weightsGen.size(); j++) { pop[i].weights.push_back(ConvertToValue<float>(pop
                [i].weightsGen[j], rangeweights)); }
174             Weight_t iWeight(pop[i].weights.begin(), pop[i].weights.begin() + ((inputneurons + 1) * hiddenneurons));
175             Weight_t hWeight(pop[i].weights.begin() + ((inputneurons + 1) * hiddenneurons), pop[i].weights.end());
176             for (uint32_t j = 0; j < inputValues.size(); j++)
177             {
178                 Predict_t results = NNfuction(inputValues[j], iWeight, hWeight, inputneurons, hiddenneurons, outputneurons);
179                 for (uint32_t k = 0; k < results.OutputNeurons.size(); k++)
180                 {
181                     pop[i].Fitness -= results.OutputNeurons[k] / goal[j].OutputNeurons[k];
182                 }
183                 pop[i].Fitness += results.OutputNeurons.size();
184             }
185             pop[i].Fitness /= inputValues.size();
186             totalFitness += pop[i].Fitness;
187         }
188     }
189
190
191     bool GA::SurvivalOfTheFittest(Population_t &pop, float &totalFitness)
192     {
193         bool retVal = false;
194         uint32_t decimationCount = pop.size() / 2;
195
196         unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
197         std::default_random_engine gen(seed);
198
```

```
199         std::sort(pop.begin(), pop.end(), PopMemberSort);
200
201         uint32_t i = ELITISME;
202         while (pop.size() > decimationCount)
203         {
204             if (i >= pop.size()) { i = ELITISME; }
205             std::uniform_real_distribution<float> dis(0, totalFitness);
206             if (dis(gen) < pop[i].Fitness)
207             {
208                 pop.erase(pop.begin() + i--);
209                 totalFitness -= pop[i].Fitness;
210             }
211             i++;
212         }
213
214         if (pop[0].Fitness < END_ERROR) { retVal = true; }
215         return retVal;
216     }
217 }
```