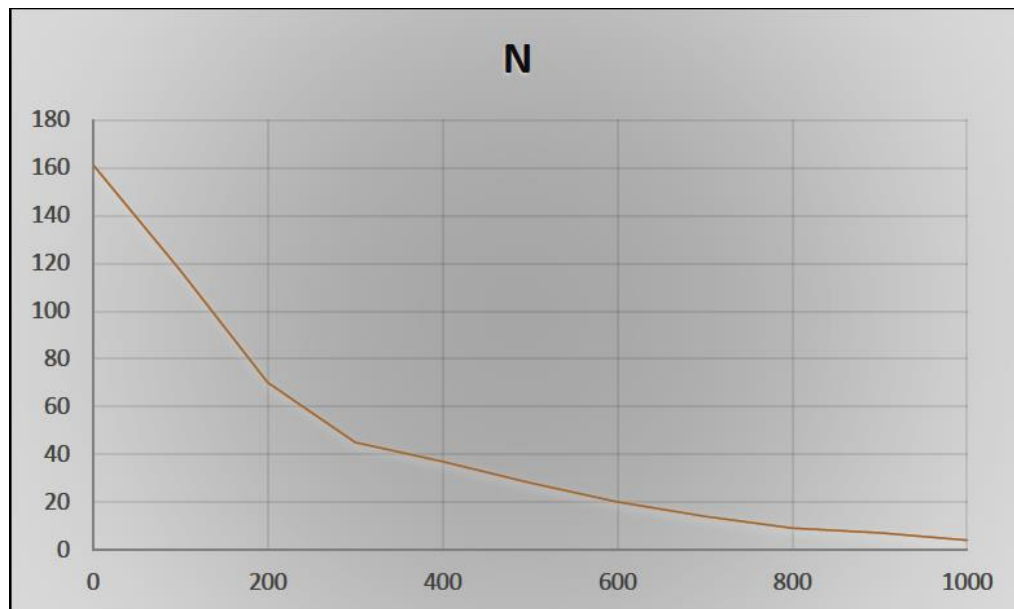


Hinton and Nowlan's Simulation of Life Time Learning

An algorithm

1. Create a population of 1000 chromosomes with 20 genes either of 1, 0, ? with probability being 0.25, 0.25, 0.5, respectively.
2. Make each chromosome a maximum of 1000 learnings by assigning "1" or "0" to each of "?"
 - * each gene feels happy or unhappy and if happy the gene is fixed and otherwise remain "?"
 - * if all "?" are happy then $N = 1000 - so.far.trials$ else keep learning till $N = 0$.
3. Select two parents at random with the probability of $1 - 19N/1000$.
4. Create next generation by repeating 3. 1000 times
5. Repeat from 2. to 4. untill solution found



Source code:

```
import java.util.*;
class Attempts
{
    public int value;
    public boolean changed;
    public Attempts()
    {
        value = 0;
        changed = false;
    }
}
class Chromosome
{
    private static int numberLearningAttempts = 1000;
    private final boolean fixGenomeWhenSuccess = false;
    private final double fitnessBoost = 19.;
```

```

private int[] genome;
private int ID;
public static void setNumberOfLearningAttempts(int numberLearningAttempts)
{
    Chromosome.numberLearningAttempts = numberLearningAttempts;
}
public Chromosome(int ID, int[] genome)
{
    this.ID = ID;
    this.genome = genome;
}
public Chromosome(int ID, int size, int fixedElements, Random random)
{
    this.ID = ID;
    genome = new int[size];
    ArrayList<Integer> positions = new ArrayList<>();
    for(int i = 0; i < size; i++)
    {
        genome[i] = 2; // init as ?
        positions.add(i);
    }
    Collections.shuffle(positions, random);
    for(int i = 0; i < fixedElements; i++)
    {
        int pos = positions.get(i);
        genome[pos] = random.nextInt(2);
    }
}
@Override
public String toString()
{
    if(genome == null)
        return "";
    return String.valueOf(ID) + '\t' + sequenceToString(genome);
}
public int[] getSequence()
{
    return this.genome;
}
public static String sequenceToString(int[] genome)
{
    if(genome == null)
        return "";
    StringBuilder s = new StringBuilder();
    for(int aGenome : genome)
        if(aGenome == 2)
            s.append('?');
        else
            s.append(aGenome);
    return s.toString();
}
public static Chromosome mate(int ID, Chromosome ind1, Chromosome ind2, Random rand)
{
    int recPos = 1 + rand.nextInt(ind1.genome.length - 1);
    int[] genomeNew = new int[ind1.genome.length];
    System.arraycopy(ind1.genome, 0, genomeNew, 0, recPos);
    System.arraycopy(ind2.genome, recPos, genomeNew, recPos, genomeNew.length - recPos);
    return new Chromosome(ID, genomeNew);
}
public static int[] getCounts(int[] genome)
{
    int[] cnts = new int[3];
    for(int aGenome : genome)

```

```

cnts[aGenome]++;
return cnts;
}
public double liveLearnAndReturnFitness(Random random, Attempts att)
{
int len = genome.length;
if(getCounts(genome)[1] == len)
// this individual has correct sequence, no learning will happen
return 1 + fitnessBoost;
else if(getCounts(genome)[0] > 0)
// this individual has 0s in the sequence, will not learn right solution
return 1;
else
{
// do learning
int[] sequence = genome.clone();
ArrayList<Integer> positionOfQuestionMarks = new ArrayList<>();
for(int i = 0; i < sequence.length; i++)
if(sequence[i] == 2)
positionOfQuestionMarks.add(i);
int attempts = 1;
for(; attempts <= numberLearningAttempts; attempts++)
{
int[] learnedBits = getRandomBinaryString(positionOfQuestionMarks.size(), random);
for(int i = 0; i < positionOfQuestionMarks.size(); i++)
sequence[positionOfQuestionMarks.get(i)] = learnedBits[i];
if(getCounts(sequence)[1] == len)
{
if(fixGenomeWhenSuccess)
genome = sequence;
att.value = attempts;
att.changed = true;
return 1 + fitnessBoost * (numberLearningAttempts - attempts) / 1000.;
}
}
// could not learn
return 1;
}
}
public static int[] getRandomBinaryString(int len, Random random)
{
int[] s = new int[len];
for(int i = 0; i < len; i++)
s[i] = random.nextBoolean() ? 1 : 0;
return s;
}
public static Chromosome sampleIndividualPropToFitness(Random random, HashMap<Chromosome,
Double> largeFitness,
ArrayList<Chromosome> fitnessOne,
double totFitnessGreaterThanOrEqualToOne)
{
double totFitness = fitnessOne.size() + totFitnessGreaterThanOrEqualToOne;
if(random.nextDouble() < fitnessOne.size() / totFitness)
// sample from those with fitness = 1
return fitnessOne.get(random.nextInt(fitnessOne.size()));
else
{
// sample from those with fitness > 1
double fitnessMark = random.nextDouble() * totFitnessGreaterThanOrEqualToOne;
double cumulativeFitness = 0.;
Chromosome sampledInd = null;
for(Chromosome ind : largeFitness.keySet())
{

```

```

cumulativeFitness += largeFitness.get(ind);
if(cumulativeFitness >= fitnessMark)
{
    sampledInd = ind;
    break;
}
}
return sampledInd;
}
}
public static boolean isFixated(ArrayList<Chromosome> inds)
{
    int[] firstGenome = inds.get(0).genome;
    for(Chromosome ind : inds)
    if(!Arrays.equals(firstGenome, ind.genome))
    return false;
    return true;
}
public static double[] getGenomeProb(Collection<Chromosome> inds)
{
    double[] res = new double[3];
    for(Chromosome ind : inds)
    {
        int[] thisInd = getCounts(ind.genome);
        res[0] += thisInd[0];
        res[1] += thisInd[1];
        res[2] += thisInd[2];
    }
    double sum = res[0] + res[1] + res[2];
    res[0] /= sum;
    res[1] /= sum;
    res[2] /= sum;
    return res;
}
}
public class Main
{
    static long seed = System.currentTimeMillis();
    public static void main(String[] args) throws InterruptedException
    {
        Random random = new Random(seed);
        int numberOfSamples = 1000;
        int len = 20;
        int fixed = 10;
        Chromosome.setNumberOfLearningAttempts(numberOfSamples);
        ArrayList<Chromosome> inds = new ArrayList<>();
        for(int i = 0; i < numberOfSamples; i++)
        inds.add(new Chromosome(i, len, fixed, random));
        int generation = 0;
        double[] populationStats = Chromosome.getGenomeProb(inds);
        while(!Chromosome.isFixated(inds))
        {
            HashMap<Chromosome, Double> largeFitness = new HashMap<>();
            ArrayList<Chromosome> fitnessOne = new ArrayList<>();
            double totFitnessGreaterThanOne = 0., totalAttempts = 0, bestAttempts = 1000;
            int cnt = 0;
            for(Chromosome ind : inds)
            {
                Attempts attempts = new Attempts();
                double fitness = ind.liveLearnAndReturnFitness(random, attempts);
                totalAttempts += attempts.value;
                if(attempts.changed)
                cnt++;
            }
        }
    }
}

```

```

if(attempts.changed && attempts.value < bestAttempts)
bestAttempts = attempts.value;
if(fitness > 1)
{
largeFitness.put(ind, fitness);
totFitnessGreaterThanOrEqualTo += fitness;
} else
fitnessOne.add(ind);
}
System.out.printf(
"GENERATION %4d\t\t0 --> %.6f\t\t1 --> %.6f\t\t? --> %.6f\t\t\taverage --> %3.6f\t\t\tbest --> %3.6f\n",
generation, populationStats[0], populationStats[1], populationStats[2], totalAttempts / cnt,
bestAttempts);
ArrayList<Chromosome> newGen = new ArrayList<>();
for(int i = 0; i < numberOfSamples; i++)
{
Chromosome samp1 = Chromosome.sampleIndividualPropToFitness(random, largeFitness, fitnessOne,
totFitnessGreaterThanOrEqualTo);
Chromosome samp2 = Chromosome.sampleIndividualPropToFitness(random, largeFitness, fitnessOne,
totFitnessGreaterThanOrEqualTo);
Chromosome newSamp = Chromosome.mate(i, samp1, samp2, random);
newGen.add(newSamp);
}
inds = newGen;
populationStats = Chromosome.getGenomeProb(inds);
generation++;
}
System.out.println("Done:\t" + Chromosome.sequenceToString(inds.get(0).getSequence()) + "\t" +
populationStats[0] +
"\t" + populationStats[1] + "\t" + populationStats[2]);
Thread.sleep(1000);
}
}}

```