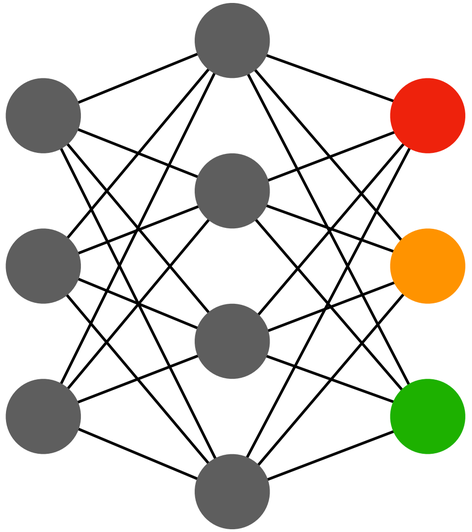
AI_TMS

The Artificial Intelligence



Samuel Arbibe 19.05.2020

The Neural Network

The neural network was built from scratch in the c ++ language. The network is completely generic and can be used in any system, complex as it might be.

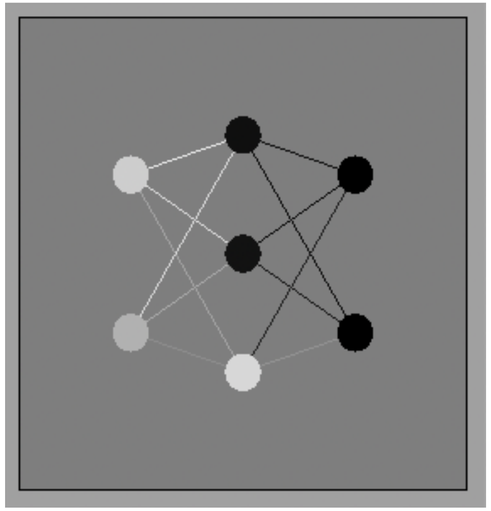
The use of the neural network is very simple, and its activity is efficient and quick.

In order to build a network, a "topology" must be created for it, and sent to the constructor function.

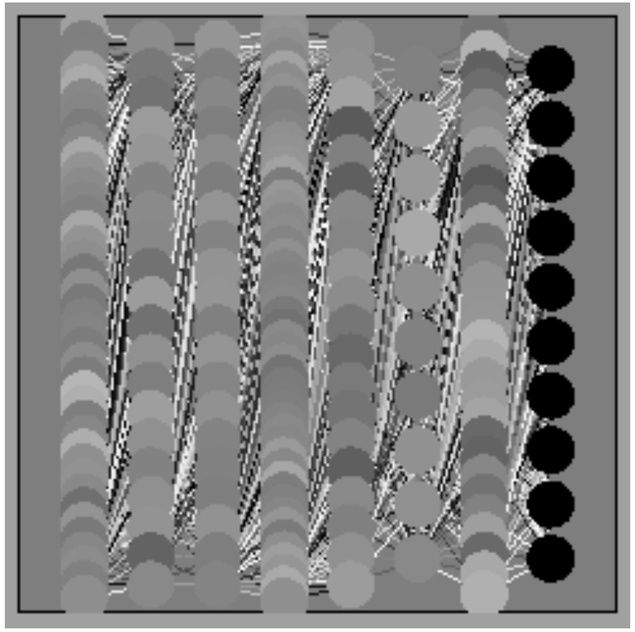
As mentioned, the network can be very simple, but also very complex.

```
vector<unsigned> topology;

// input neurons : max lane density, max queue length
topology.push_back(2);
// hidden neurons
topology.push_back(3);
// output neurons : priority points, phase time
topology.push_back(2);
```



```
// input neurons
topology.push_back(40);
// hidden neurons
topology.push_back(20);
topology.push_back(20);
topology.push_back(50);
topology.push_back(20);
topology.push_back(10);
topology.push_back(30);
// output neurons
topology.push_back(10);
```



The Genetic Algorithm

The genetic algorithm is based on the principle of ‘survival of the fittest', and aims to preserve strong 'genetic' traits and omit weak traits.

The competence of a neuronal network is determined by its performance in the simulation it controls.

The higher the traffic flow in the simulation, the better the network that set the policy at the time.

First, a generation is defined as an array of neural networks. The first generation is completely randomised.

```
vector<Net> Net::Generation = vector<Net>();
```

```
for(unsigned i = 0; i < Net::PopulationSize; i++)
{
    Net::Generation.emplace_back(topology);
}
```

```
Net::CurrentNet = &(Net::Generation[Net::CurrentNetIndex]);
```

Random generation of neural networks is done by placing random values in the weight of the neural networks.

```
// create random output weights
for (int c = 0; c < numOutputs; c++)
{
    // push a random into the output weights
    output_weights_.push_back(Connection());
    output_weights_.back().weight = randomize_weight();
}
```

```
static double randomize_weight() { return random() / double(RAND_MAX); }
```

After running an entire generation of random networks, each network is given a score.

We'll normalise the network scores: Each network in the array will receive a property called “Fitness”.

This property will express the net’s share of the grade in the sum of grades of the generation.

Fitness is exponential in relation to score, in order to increase the significance of small differences at the upper end of the score range.

The total fitness of all networks in the generation is 1, we will use this fact below.

```
void Net::NormalizeFitness(vector<Net> &oldGen) {
    double sum = 0;
    for(unsigned i = 0; i < oldGen.size(); i++)
    {
        double score = pow(oldGen[i].score_, 2);
        oldGen[i].score_ = score;
        sum += score;
    }

    for(unsigned i = 0; i < oldGen.size(); i++)
    {
        oldGen[i].fitness_ = oldGen[i].score_ / sum;
    }
}
```

After we normalize the network generation and determine the “fitness” of each of the networks, we will create the next generation and perform an "evolution".

The new generation will be based on the previous one:

When creating a new generation, a network from the previous generation is semi-randomaly selected and is replicated into the new generation.

```
vector<Net> Net::Generate(const vector<Net> &oldGen) {
    vector<Net> newGen;
    for(unsigned i = 0; i < oldGen.size(); i++)
    {
        newGen.push_back(Net::PoolSelection(oldGen));
        newGen.back().mutate(0.2);
    }
    return newGen;
}
```

The Selection Algorithm, or PoolSelection, is a random selection algorithm that is affected by a network's capabilities.

This means that for some random value, the chance of a network being chosen is equal to its fitness.

How it works:

We will create a random value **r** between 0 and 1.

As long as **r** > 0, we will lower the fitness of another network every time, in a sequential order.

When **r** is lower than 0 after subtracting some network fitness, that network is selected.

This implements the idea that each network receives a 'share' in the range of 0 to 1, depending on its fitness. The larger the share, the more likely it is to include a random value.

```
Net Net::PoolSelection(const vector<Net> &oldGen) {
    unsigned index = 0;
    double r = rand() / double(RAND_MAX);

    while(r > 0)
    {
        r -= oldGen[index].fitness_;
        index++;
    }
    index--;
    return oldGen[index];
}
```

Once we have selected networks to replicate to the new generation using the algorithm, we will mutate them.

The mutation, for a given value X, would make random changes in each neural network:

For every weight on the net, there is a X chance of it being randomly redefined.

Once the new generation is created, we will run it, and repeat.

Implementation

The role of a network in each simulation is to determine the traffic light policy at each intersection.

Each phase cycle contains an array of phases. A phase contains a number of lanes that are not necessarily on the same road, but will open and close according to the state of the phase.

We will sort this array and keep it sorted at all times, according to the phase priority. The phase that opens will be the last phase of the array, which is has highest priority. When it closes, the subsequent phase in the array, that will have the highest priority at the time, will open.

As mentioned, phase priority is calculated at every moment, so changes in traffic are reflected in real-time in the phase policies.

```
void Cycle::cycle_phases() {
    // if cycle has a minimum of 2 phases
    if (number_of_phases_ >= 2)
    {
        // when current phase is closed, advance to next phase and open it
        if (!phases_.back()->GetIsOpen())
        {
            Phase *backPhase = phases_[number_of_phases_ - 1];
            phases_[number_of_phases_ - 1] = phases_[number_of_phases_ - 2];
            phases_[number_of_phases_ - 2] = backPhase;

            phases_[number_of_phases_ - 1]->Open();

        }
            // constantly sort the list by their priority score
        else
        {
            // calculate the priority of each phase
            calculate_priority();
            // sort(arr[0:-2])
            partial_sort(phases_.begin(),
                         phases_.end() - 1,
                         phases_.end() - 1,
                         compare_priority);
        }
    }
}
```

In order to calculate phase priority, we will use the neural network.

For every phase, the neural network receives as **input**:

1. The **maximum density** value of all lanes belonging to the phase.
2. The **maximum queue length** of all lanes belonging to the phase.

Then, the neural network will return as **output**:

1. The **priority score** of the phase
2. The **optimal opening duration** of the phase

```
void Cycle::calculate_priority() {

    for (int p = 0; p < number_of_phases_ - 1; p++)
    {
        // get input values
        phases_[p]->GetInputValues(input_values_);
        // feed data through nerual net
        Net::CurrentNet->FeedForward(input_values_);
        // get data from output layer
        Net::CurrentNet->GetResults(output_values_);

        // set the output data
        phases_[p]->SetPhasePriority(output_values_[0]);
        phases_[p]->SetCycleTime(clamp(
            float(output_values_[1]) * Settings::MaxCycleTime,
            Settings::MinCycleTime,
            Settings::MaxCycleTime));
    }
}
```

At all times, a copy of the highest-rated network to date is being kept.

In the UI, there is the option to run the most best network.

It is also possible to continue to run the genetic algorithm and continue evolution all the time without interruption.

The evolution does not fixate the neural net to a certain shape.

This means that the algorithm is capable of making drastic changes to the evolution of the networks according to a variable state of the traffic, at any time.

For example, in a rainy state where cars will travel more slowly, the algorithm will know how to change the evolution accordingly, and may change the form of the network and policy over a relatively short period of time.

Source Code - https://github.com/samuelarbibe/AI_TMS

Video Demonstration - https://www.youtube.com/watch?v=BLz_PdU2oyo

Video Explanation - https://www.youtube.com/watch?v=xJOcDKXWJXo