TARAS SHEVCHENKO

NATIONAL UNIVERSITY OF KYIV

Faculty of Computer Science and Cybernetics

Department of Mathematical Informatics

TERM WORK

Specialization: 122 Computer Science

Training direction: “Artificial Intelligence”

on the topic:

DEVELOPMENT OF THE AI GAME BOT USING NEURAL NETWORKS AND GENETIC ALGORITHM

Made by 1-st year master student

Zubko Zinovii Vasylovych ____________________

(signature)

Academic advisor:

Ph.D., Associate Professor:

Panchenko Taras Volodymyrovych ____________________

(signature)

I certify that in this work there are no borrowings from the works of other authors without the corresponding references.

Student ____________________

(signature)

Kyiv – 2019

CONTENT

Introduction

Neural networks and genetic algorithms demonstrate powerful problem solving ability. They are based on quite simple principles, but take advantage of their mathematical nature: non-linear iteration. Neural networks with backpropagation learning showed results by searching for various kinds of functions. However, the choice of the basic parameter (network topology, learning rate, initial weights) often already determines the success of the training process. The selection of these parameter follow in practical use rules of thumb, but their value is at most arguable.

Genetic algorithms are global search methods, that are based on principles like selection, crossover and mutation. This course work examines how genetic algorithms can be used for improving Neural Network which is used for AI bot, which plays with another implemented algorithms.

Building the perfect deep learning network involves a hefty amount of art to accompany sound science. One way to go about finding the right hyperparameters is through brute force trial and error: Try every combination of sensible parameters, send them to your Spark cluster, go about your daily jive, and come back when you have an answer.

But there's gotta be a better way!
Here, we try to improve upon the brute force method by applying a genetic algorithm to evolve a network with the goal of achieving optimal hyperparameters in a fraction the time of a brute force search.

How much faster?
Let's say it takes five minutes to train and evaluate a network on your dataset. And let's say we have four parameters with five possible settings each. To try them all would take (5**4) * 5 minutes, or 3,125 minutes, or about 52 hours.

Now let's say we use a genetic algorithm to evolve 10 generations with a population of 20 (more on what this means below), with a plan to keep the top 25% plus a few more, so ~8 per generation. This means that in our first generation we score 20 networks (20 * 5 = 100 minutes). Every generation after that only requires around 12 runs, since we don't have the score the ones we keep. That's 100 + (9 generations * 5 minutes * 12 networks) = 640 minutes, or 11 hours.

We've just reduced our parameter tuning time by almost 80%! That is, assuming it finds the best parameters…

NNs have helped us solve so many problems. But there's a huge problem that they still have. Hyperparameters! These are the only values that can not be learned… Until now.

Note: Hyper-parameters are values required by the NN to perform properly, given a problem.
We can use GAs to learn the best hyper-parameters for a NN! This is absolutely awesome!

Now, we don't have to worry about "knowing the right hyperparameters" since, they can be learned using a GA. Also, we can use this to learn the parameter's(weights) of a NN as well.

Since, in a GA, the entities learn the optimum genome for the specified problem, here, the genome of each NN will be its set of hyper-parameters.

Neural Network

An Artificial Neural Network (ANN) is an information processing paradigm that is inspired by the way biological nervous systems, such as the brain, process information. The key element of this paradigm is the novel structure of the information processing system. It is composed of a large number of highly interconnected processing elements (neurones) working in unison to solve specific problems. ANNs, like people, learn by example. An ANN is configured for a specific application, such as pattern recognition or data classification, through a learning process. Learning in biological systems involves adjustments to the synaptic connections that exist between the neurones. This is true of ANNs as well.

Why use neural networks?

Neural networks, with their remarkable ability to derive meaning from complicated or imprecise data, can be used to extract patterns and detect trends that are too complex to be noticed by either humans or other computer techniques. A trained neural network can be thought of as an "expert" in the category of information it has been given to analyse. This expert can then be used to provide projections given new situations of interest and answer "what if" questions.
Other advantages include:

1. Adaptive learning: An ability to learn how to do tasks based on the data given for training or initial experience.
2. Self-Organisation: An ANN can create its own organisation or representation of the information it receives during learning time.
3. Real Time Operation: ANN computations may be carried out in parallel, and special hardware devices are being designed and manufactured which take advantage of this capability.
4. Fault Tolerance via Redundant Information Coding: Partial destruction of a network leads to the corresponding degradation of performance. However, some network capabilities may be retained even with major network damage.

Neural networks versus conventional computers

Neural networks take a different approach to problem solving than that of conventional computers. Conventional computers use an algorithmic approach i.e. the computer follows a set of instructions in order to solve a problem. Unless the specific steps that the computer needs to follow are known the computer cannot solve the problem. That restricts the problem solving capability of conventional computers to problems that we already understand and know how to solve. But computers would be so much more useful if they could do things that we don't exactly know how to do.

Neural networks process information in a similar way the human brain does. The network is composed of a large number of highly interconnected processing elements(neurones) working in parallel to solve a specific problem. Neural networks learn by example. They cannot be programmed to perform a specific task. The examples must be selected carefully otherwise useful time is wasted or even worse the network might be functioning incorrectly. The disadvantage is that because the network finds out how to solve the problem by itself, its operation can be unpredictable.

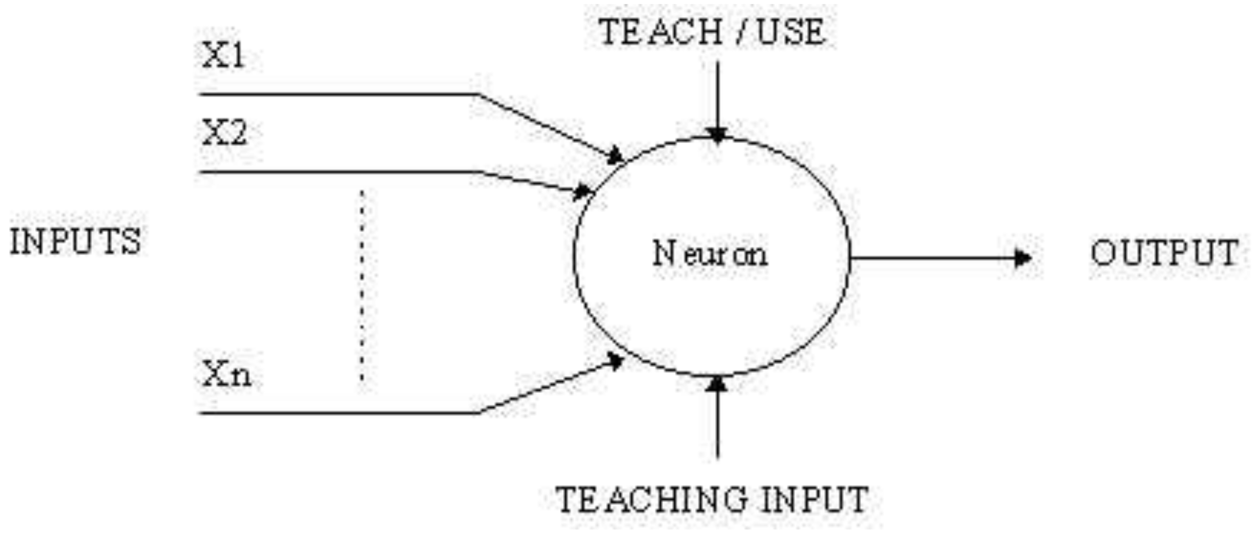
On the other hand, conventional computers use a cognitive approach to problem solving; the way the problem is to solved must be known and stated in small unambiguous instructions. These instructions are then converted to a high level language program and then into machine code that the computer can understand. These machines are totally predictable; if anything goes wrong is due to a software or hardware fault.

Neural networks and conventional algorithmic computers are not in competition but complement each other. There are tasks are more suited to an algorithmic approach like arithmetic operations and tasks that are more suited to neural networks. Even more, a large number of tasks, require systems that use a combination of the two approaches (normally a conventional computer is used to supervise the neural network) in order to perform at maximum efficiency.

Neural networks do not perform miracles. But if used sensibly they can produce some amazing results.

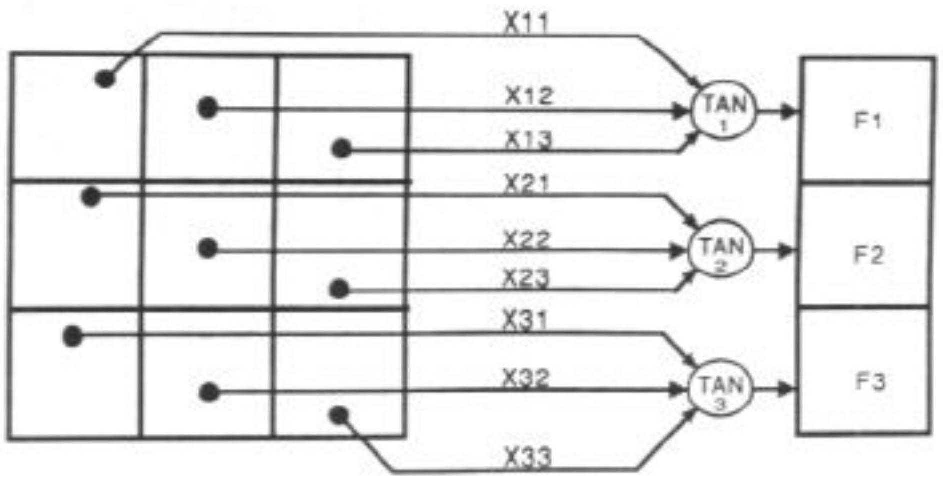
A simple neuron

An artificial neuron is a device with many inputs and one output. The neuron has two modes of operation; the training mode and the using mode. In the training mode, the neuron can be trained to fire (or not), for particular input patterns. In the using mode, when a taught input pattern is detected at the input, its associated output becomes the current output. If the input pattern does not belong in the taught list of input patterns, the firing rule is used to determine whether to fire or not.



Pattern Recognition

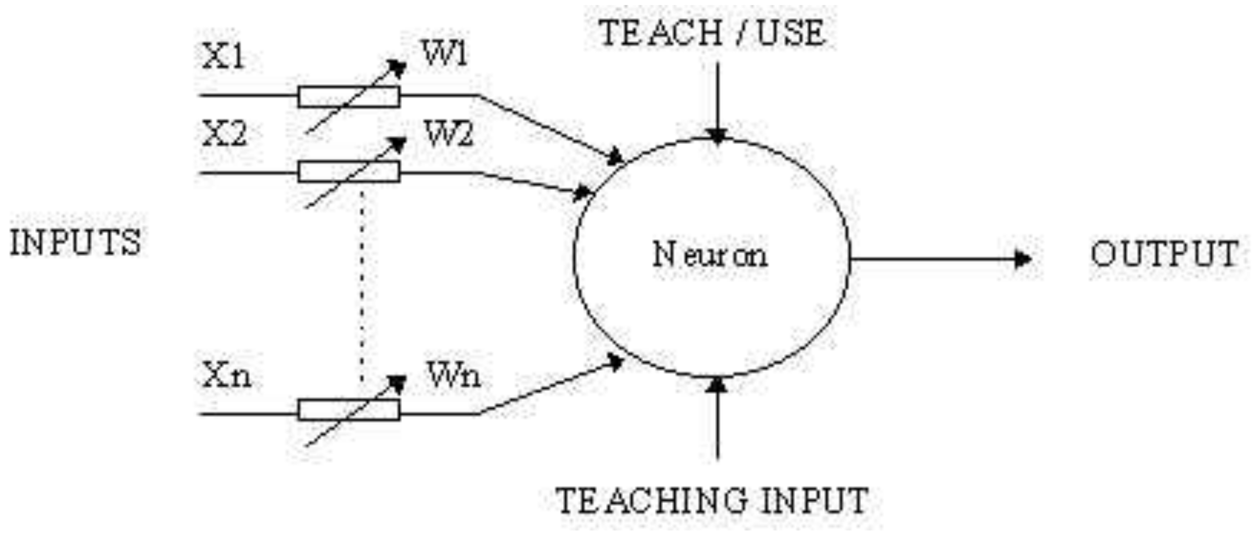
An important application of neural networks is pattern recognition. Pattern recognition can be implemented by using a feed-forward (figure 1) neural network that has been trained accordingly. During training, the network is trained to associate outputs with input patterns. When the network is used, it identifies the input pattern and tries to output the associated output pattern. The power of neural networks comes to life when a pattern that has no output associated with it, is given as an input. In this case, the network gives the output that corresponds to a taught input pattern that is least different from the given pattern.



A

more complicated neuron

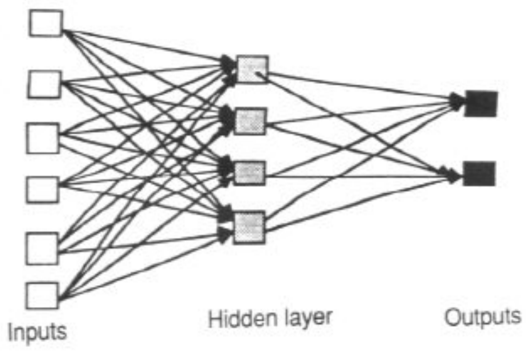
The previous neuron doesn't do anything that conventional conventional computers don't do already. A more sophisticated neuron (figure 2) is the McCulloch and Pitts model (MCP). The difference from the previous model is that the inputs are 'weighted', the effect that each input has at decision making is dependent on the weight of the particular input. The weight of an input is a number which when multiplied with the input gives the weighted input. These weighted inputs are then added together and if they exceed a pre-set threshold value, the neuron fires. In any other case the neuron does not fire.



Architecture of neural networks

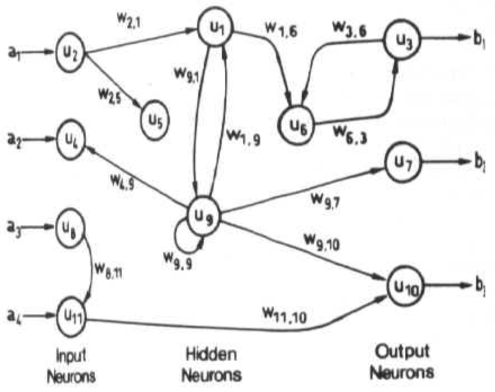
Feed-forward networks

Feed-forward ANNs allow signals to travel one way only; from input to output. There is no feedback (loops) i.e. the output of any layer does not affect that same layer. Feed-forward ANNs tend to be straight forward networks that associate inputs with outputs. They are extensively used in pattern recognition. This type of organisation is also referred to as bottom-up or top-down.



Feedback networks

Feedback networks can have signals travelling in both directions by introducing loops in the network. Feedback networks are very powerful and can get extremely complicated. Feedback networks are dynamic; their 'state' is changing continuously until they reach an equilibrium point. They remain at the equilibrium point until the input changes and a new equilibrium needs to be found. Feedback architectures are also referred to as interactive or recurrent, although the latter term is often used to denote feedback connections in single-layer organisations.



Network layers

The commonest type of artificial neural network consists of three groups, or layers, of units: a layer of "**input**" units is connected to a layer of "**hidden**" units, which is connected to a layer of "**output**" units. (see Figure 4.1)

- The activity of the input units represents the raw information that is fed into the network.

- The activity of each hidden unit is determined by the activities of the input units and the weights on the connections between the input and the hidden units.

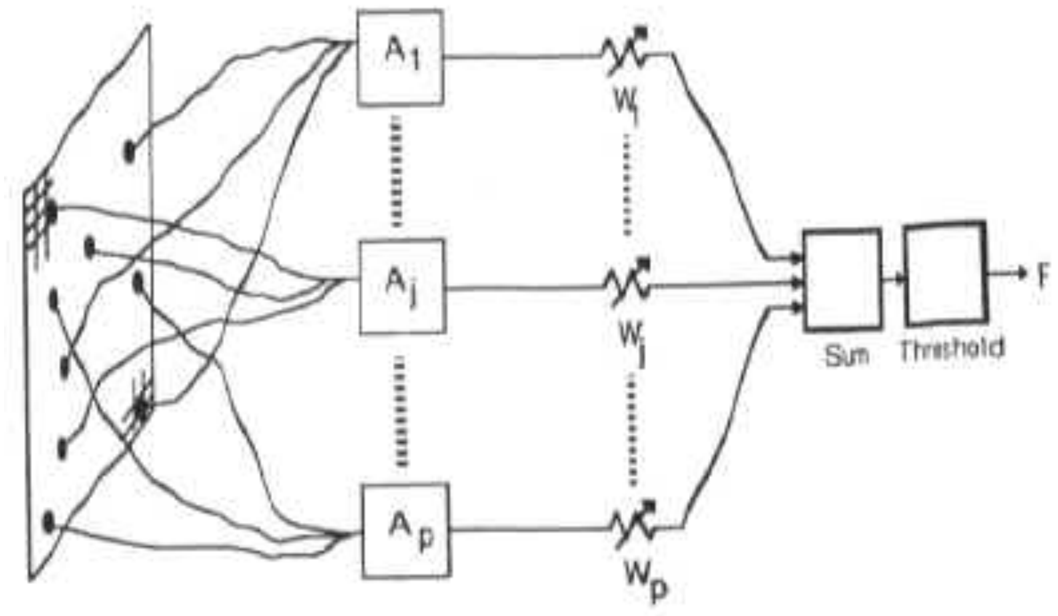
● The behaviour of the output units depends on the activity of the hidden units and the weights between the hidden and output units.

This simple type of network is interesting because the hidden units are free to construct their own representations of the input. The weights between the input and hidden units determine when each hidden unit is active, and so by modifying these weights, a hidden unit can choose what it represents.

We also distinguish single-layer and multi-layer architectures. The single-layer organisation, in which all units are connected to one another, constitutes the most general case and is of more potential computational power than hierarchically structured multi-layer organisations. In multi-layer networks, units are often numbered by layer, instead of following a global numbering.

Perceptrons

The most influential work on neural nets in the 60's went under the heading of 'perceptrons' a term coined by Frank Rosenblatt. The perceptron (figure 4.4) turns out to be an MCP model (neuron with weighted inputs) with some additional, fixed, pre--processing. Units labelled A1, A2, Aj , Ap are called association units and their task is to extract specific, localised featured from the input images. Perceptrons mimic the basic idea behind the mammalian visual system. They were mainly used in pattern recognition even though their capabilities extended a lot more.



In 1969 Minsky and Papert wrote a book in which they described the limitations of single layer Perceptrons. The impact that the book had was tremendous and caused a lot of neural network researchers to loose their interest. The book was very well written and showed mathematically that *single layer* perceptrons could not do some basic pattern recognition operations like determining the parity of a shape or determining whether a shape is connected or not. What they did not realised, until the 80's, is that given the appropriate training, multilevel perceptrons can do these operations.

The Learning Process of ANN

The memorisation of patterns and the subsequent response of the network can be categorised into two general paradigms:

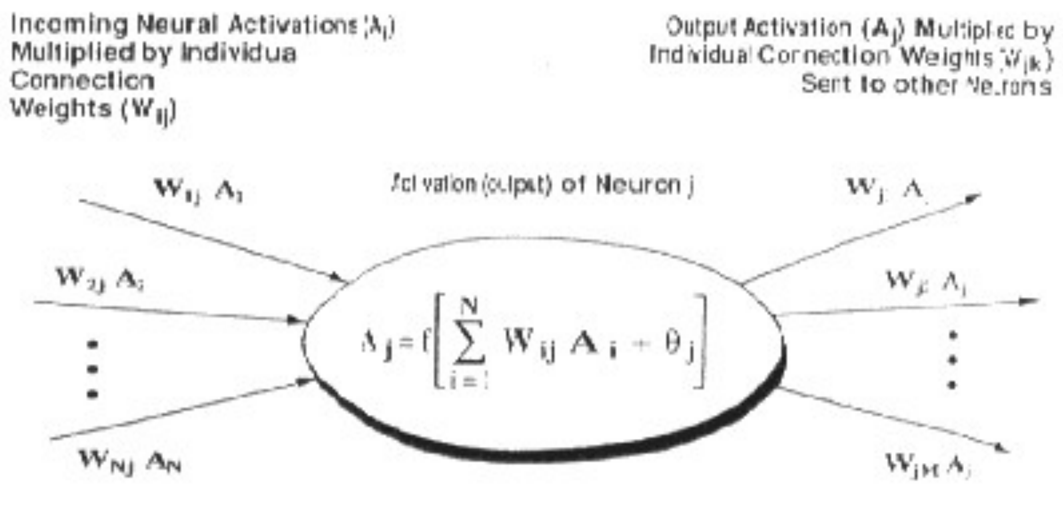
- **associative mapping** in which the network learns to produce a particular pattern on the set of input units whenever another particular pattern is applied on the set of input units. The associtive mapping can generally be broken down into two mechanisms:

  - *auto-association*: an input pattern is associated with itself and the states of input and output units coincide. This is used to provide pattern completition, ie to produce a pattern whenever a portion of it or a distorted pattern is presented. In the second case, the network actually stores pairs of patterns building an association between two sets of patterns.
  - *hetero-association*: is related to two recall mechanisms:

    - *nearest-neighbour* recall, where the output pattern produced corresponds to the input pattern stored, which is closest to the pattern presented, and
    - *interpolative* recall, where the output pattern is a similarity dependent interpolation of the patterns stored corresponding to the pattern presented. Yet another paradigm, which is a variant associative mapping is classification, ie when there is a fixed set of categories into which the input patterns are to be classified.

- **regularity detection** in which units learn to respond to particular properties of the input patterns. Whereas in asssociative mapping the network stores the relationships among patterns, in regularity detection the response of each unit has a particular 'meaning'. This type of learning mechanism is essential for feature discovery and knowledge representation.

Every neural network posseses knowledge which is contained in the values of the connections weights. Modifying the knowledge stored in the network as a function of experience implies a learning rule for changing the values of the weights.


Incoming Neural Activations (A_i) Multiplied by Individual Connection Weights (W_ij)
Output Activation (A_j) Multiplied by Individual Connection Weights (W_jk) Sent to other Neurons
Activation (output) of Neuron j
W_1j A_1
W_2j A_2
W_Nj A_N
W_j1 A_j
W_j2 A_j
W_jM A_j
A_j = f[Σ_{i=1}^{N} W_ij A_i + θ_j]

Information is stored in the weight matrix W of a neural network. Learning is the determination of the weights. Following the way learning is performed, we can distinguish two major categories of neural networks:

- **fixed networks** in which the weights cannot be changed, ie dW/dt=0. In such networks, the weights are fixed a priori according to the problem to solve.

- **adaptive networks** which are able to change their weights, ie dW/dt not= 0.

All learning methods used for adaptive neural networks can be classified into two major categories:

- **Supervised learning** which incorporates an external teacher, so that each output unit is told what its desired response to input signals ought to be. During the learning process global information may be required. Paradigms of supervised learning include error-correction learning, reinforcement learning and stochastic learning.
  An important issue conserning supervised learning is the problem of error convergence, ie the minimisation of error between the desired and computed unit values. The aim is to determine a set of weights which minimises the error. One well-known method, which is common to many learning paradigms is the least mean square (LMS) convergence.

- **Unsupervised learning** uses no external teacher and is based upon only local information. It is also referred to as self-organisation, in the sense that it self-organises data presented to the network and detects their emergent collective properties. Paradigms of unsupervised learning are Hebbian lerning and competitive learning.
  Ano2.2 From Human Neurones to Artificial Neuronesther aspect of learning concerns the distinction or not of a seperate phase, during which the network is trained, and a subsequent operation phase. We say that a neural network learns off-line if the learning phase and the operation phase are distinct. A neural network learns on-line if it learns and operates at the same time. Usually, supervised learning is performed off-line, whereas usupervised learning is performed on-line.

Transfer Function

The behaviour of an ANN (Artificial Neural Network) depends on both the weights and the input-output function (transfer function) that is specified for the units. This function typically falls into one of three categories:

- linear (or ramp)

- threshold

- sigmoid

For **linear units**, the output activity is proportional to the total weighted output.

For **threshold units**, the output is set at one of two levels, depending on whether the total input is greater than or less than some threshold value.

For **sigmoid units**, the output varies continuously but not linearly as the input changes. Sigmoid units bear a greater resemblance to real neurones than do linear or threshold units, but all three must be considered rough approximations.

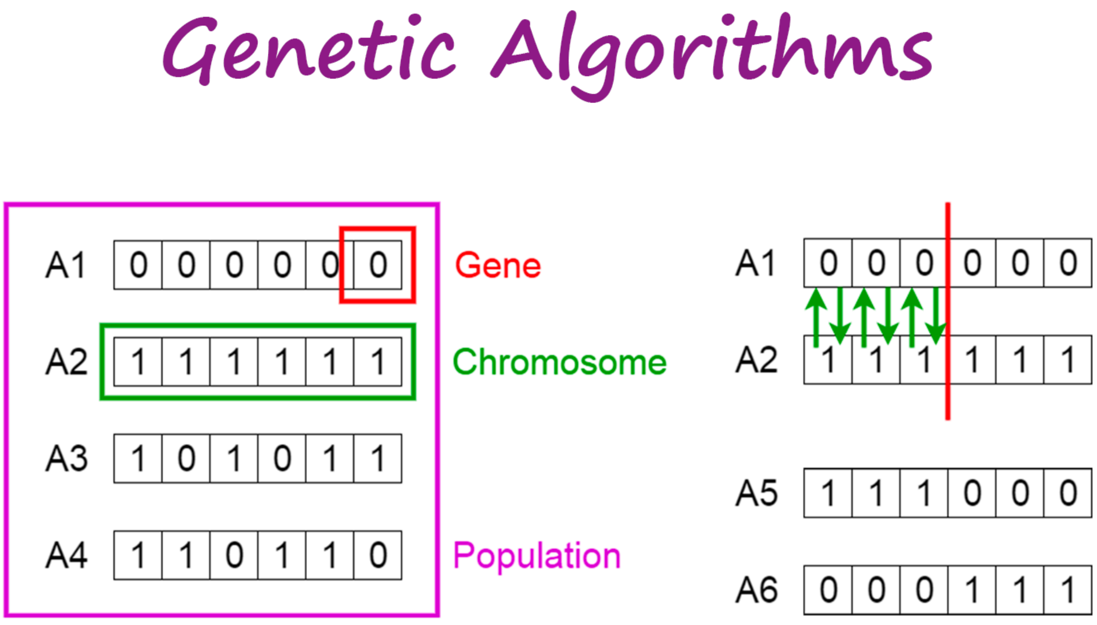
To make a neural network that performs some specific task, we must choose how the units are connected to one another (see figure 4.1), and we must set the weights on the connections appropriately. The connections determine whether it is possible for one unit to influence another. The weights specify the strength of the influence.

We can teach a three-layer network to perform a particular task by using the following procedure:

1. We present the network with training examples, which consist of a pattern of activities for the input units together with the desired pattern of activities for the output units.
2. We determine how closely the actual output of the network matches the desired output.
3. We change the weight of each connection so that the network produces a better approximation of the desired output.

GENETIC ALGORITHM

A **genetic algorithm** is a search heuristic that is inspired by Charles Darwin's theory of natural evolution. This algorithm reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation.



Notion of Natural Selection

The process of natural selection starts with the selection of fittest individuals from a population. They produce offspring which inherit the characteristics of the parents and will be added to the next generation. If parents have better fitness, their offspring will be better than parents and have a better chance at surviving. This process keeps on iterating and at the end, a generation with the fittest individuals will be found.
This notion can be applied for a search problem. We consider a set of solutions for a problem and select the set of best ones out of them.

Five phases are considered in a genetic algorithm.

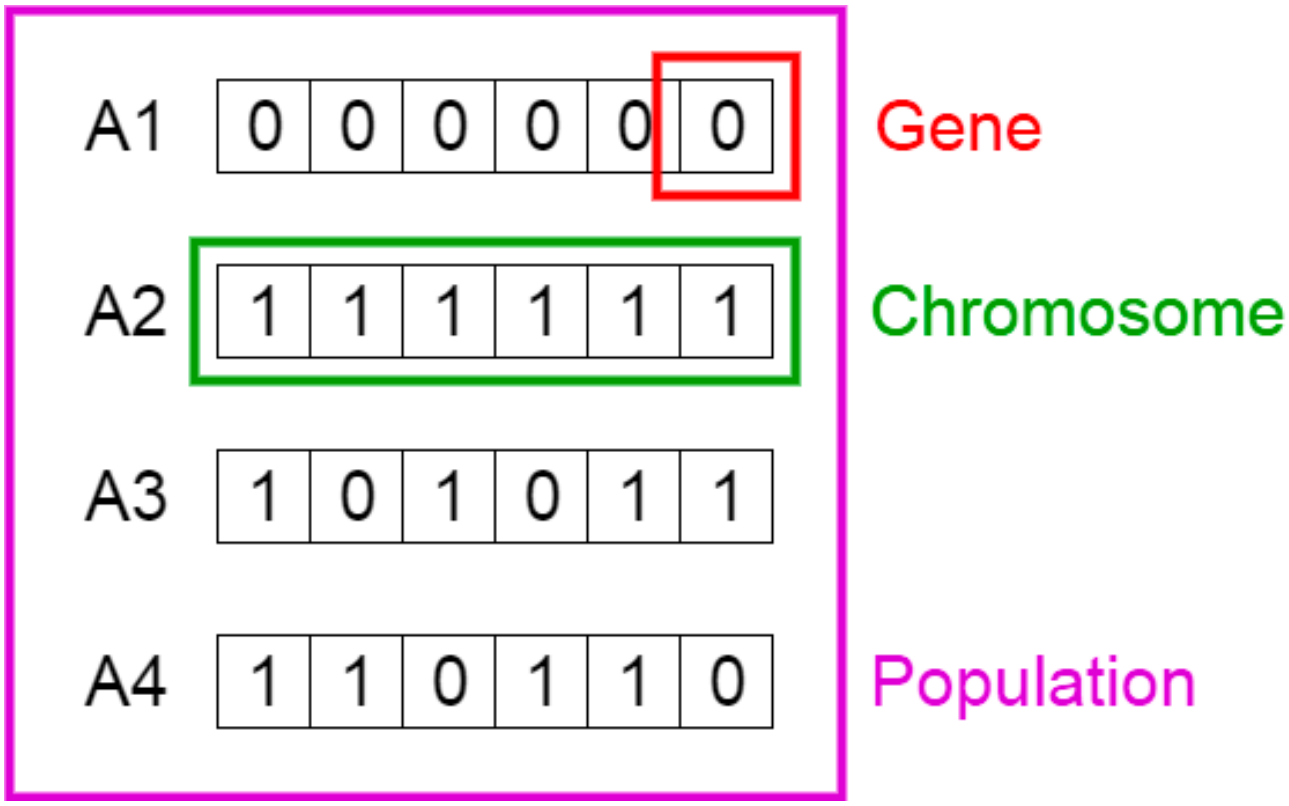
1. Initial population
2. Fitness function
3. Selection
4. Crossover
5. Mutation

Initial Population

The process begins with a set of individuals which is called a **Population**. Each individual is a solution to the problem you want to solve.

An individual is characterized by a set of parameters (variables) known as **Genes**. Genes are joined into a string to form a **Chromosome** (solution).

In a genetic algorithm, the set of genes of an individual is represented using a string, in terms of an alphabet. Usually, binary values are used (string of 1s and 0s). We say that we encode the genes in a chromosome.



Fitness Function

The **fitness function** determines how fit an individual is (the ability of an individual to compete with other individuals). It gives a fitness score to each individual. The probability that an individual will be selected for reproduction is based on its **fitness score**.

Selection

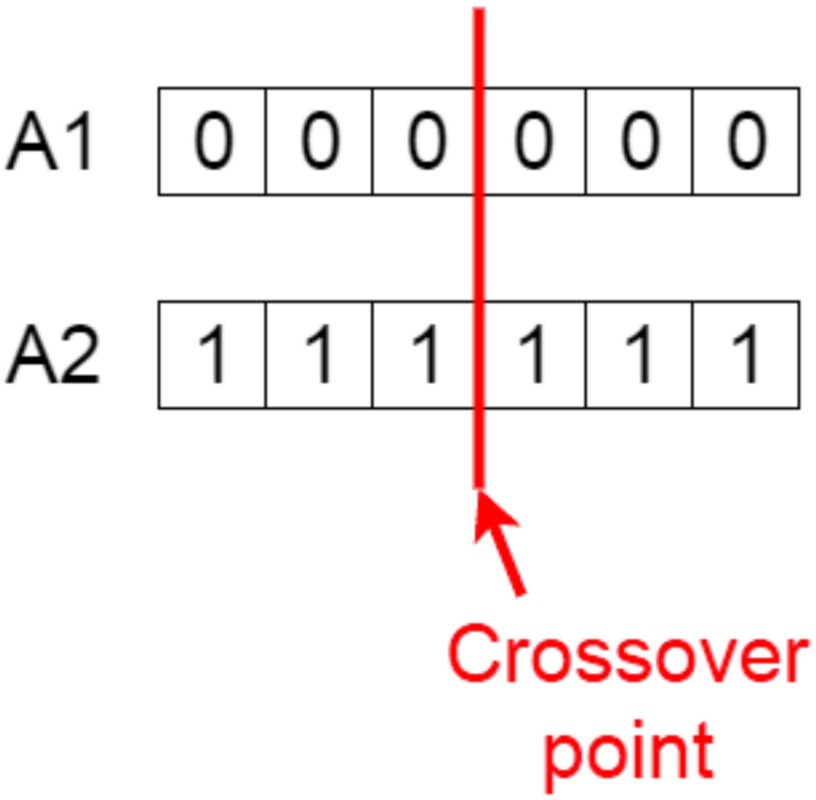
The idea of **selection** phase is to select the fittest individuals and let them pass their genes to the next generation.

Two pairs of individuals (**parents**) are selected based on their fitness scores. Individuals with high fitness have more chance to be selected for reproduction.

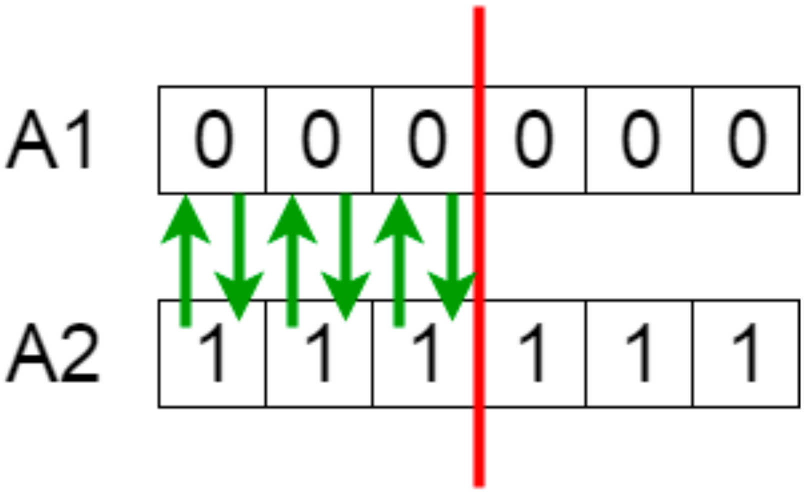
Crossover

Crossover is the most significant phase in a genetic algorithm. For each pair of parents to be mated, a **crossover point** is chosen at random from within the genes.

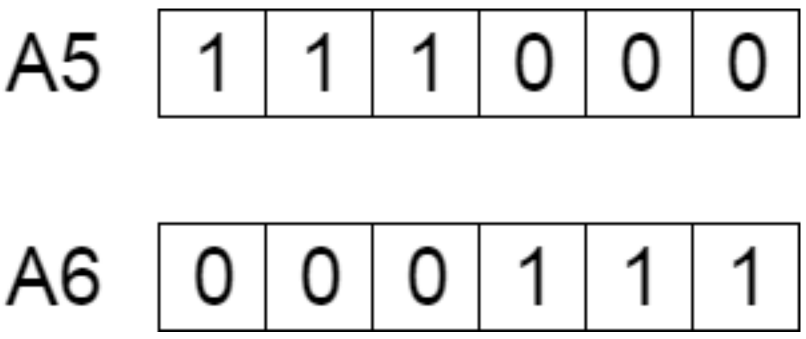
For example, consider the crossover point to be 3 as shown below.



Offspring are created by exchanging the genes of parents among themselves until the crossover point is reached.

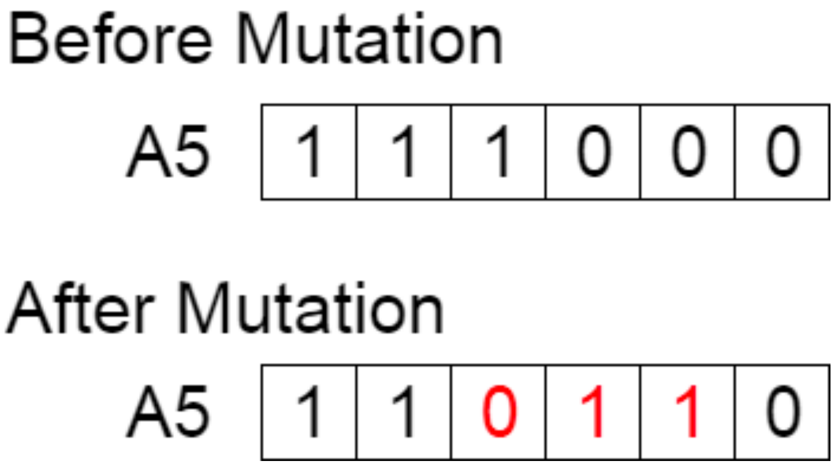


The new offspring are added to the population.



Mutation

In certain new offspring formed, some of their genes can be subjected to a **mutation** with a low random probability. This implies that some of the bits in the bit string can be flipped.



Mutation occurs to maintain diversity within the population and prevent premature convergence.

Termination

The algorithm terminates if the population has converged (does not produce offspring which are significantly different from the previous generation). Then it is said that the genetic algorithm has provided a set of solutions to our problem.

Comments

The population has a fixed size. As new generations are formed, individuals with least fitness die, providing space for new offspring.
The sequence of phases is repeated to produce individuals in each new generation which are better than the previous generation.

Psuedocode

```
START
Generate the initial population
Compute fitness
REPEAT
    Selection
    Crossover
    Mutation
    Compute fitness
UNTIL population has converged
STOP
```

Applying genetic algorithms to Neural Networks

We'll attempt to evolve a fully connected network (MLP). Our goal is to find the best parameters for an image classification task.
We'll tune four parameters:

- Number of layers (or the network depth)
- Neurons per layer (or the network width)
- Dense layer activation function
- Network optimizer

The steps we'll take to evolve the network, similar to those described above, are:

1. Initialize *N* random networks to create our population.
2. Score each network. This takes some time: We have to train the weights of each network and then see how well it performs at classifying the test set. Since this will be an image classification task, we'll use classification accuracy as our fitness function.
3. Sort all the networks in our population by score (accuracy). We'll keep some percentage of the top networks to become part of the next generation and to breed children.
4. We'll also randomly keep a few of the non-top networks. This helps find potentially lucky combinations between worse-performers and top performers, and also helps keep us from getting stuck in a local maximum.
5. Now that we've decided which networks to keep, we randomly mutate some of the parameters on some of the networks.
6. Here comes the fun part: Let's say we started with a population of 20 networks, we kept the top 25% (5 nets), randomly kept 3 more loser networks, and mutated a few of them. We let the other 12 networks die. In an effort to keep our population at 20 networks, we need to fill 12 open spots. It's time to breed!

Breeding

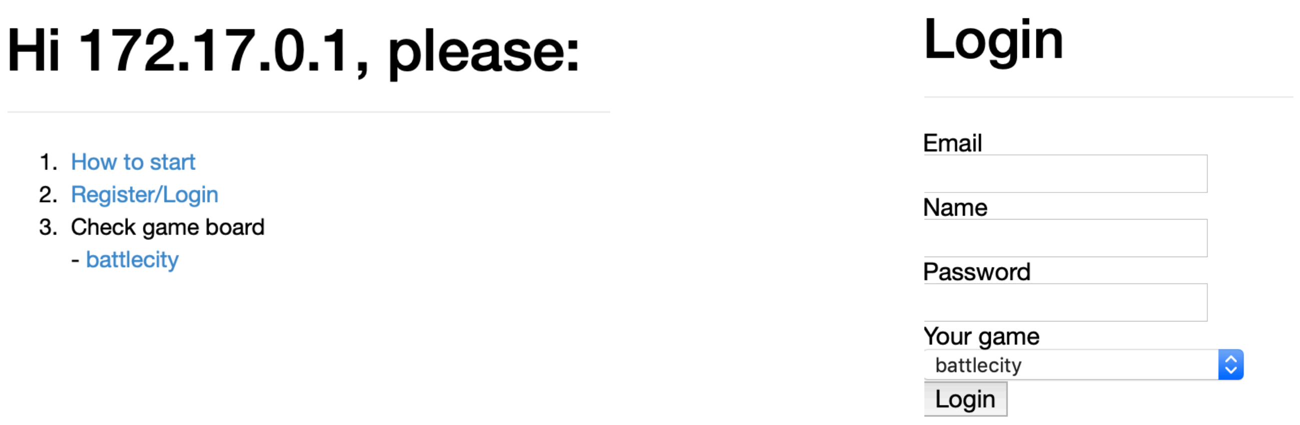
Breeding is where we take two members of a population and generate one or more child, where that child represents a combination of its parents.
In our neural network case, each child is a combination of a random assortment of parameters from its parents. For instance, one child might have the same number of layers as its mother and the rest of its parameters from its father. A second child of the same parents may have the opposite. You can see how this mirrors real-world biology and how it can lead to an optimized network quickly.

Task - AI tank bot

The task is to write your AI bot for a tanchik who will beat other bots on points. The whole game takes place on the same field. Tanchik can move on free cells in all four directions. The same tank can shoot a projectile, which will explode if it hits an obstacle. The projectile moves faster than the tank twice.

For the murder of enemies, the bot of the player earns points. Penalty points are awarded for the death of a tank. Points are added up. The winner is a player with a large number of points (before the agreed time). Dead tank immediately appears in a random place on the field.

So, the player is registered on the server, indicating his/her **email**



Hi 172.17.0.1, please:

1. How to start
2. Register/Login
3. Check game board
   - battlecity

Login

Email

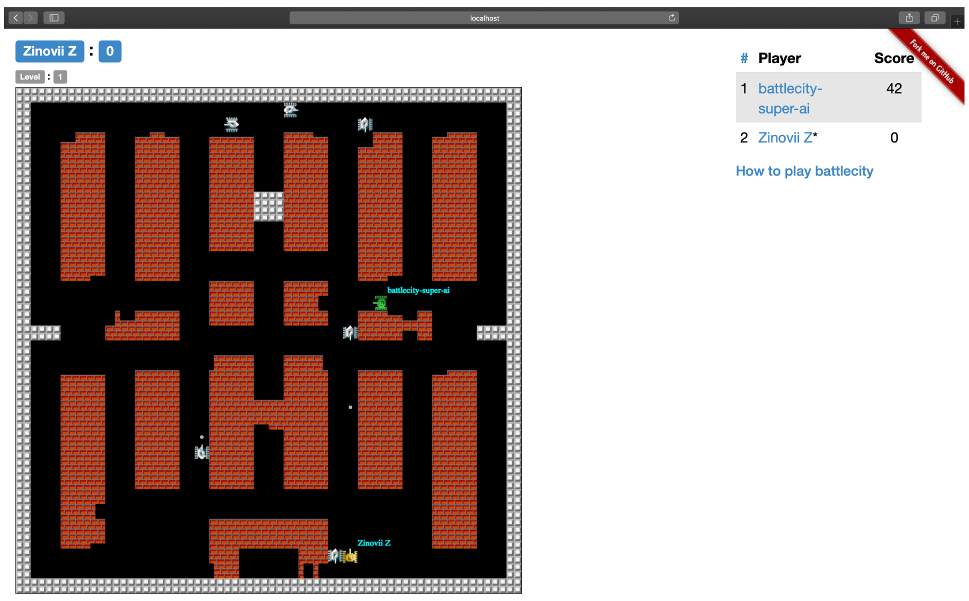
Name

Password

Your game
battlecity

Login

Next, you need to connect from the code to the server via web sockets. Address to connect to the game on a server deployed on the local network:
ws://server_ip:8080/codenjoy-contest/wsuser=your@email.com&code=12345678901234567890



#	Player	Score
1	battlecity-super-ai	42
2	Zinovii Z*	0

How to play battlecity

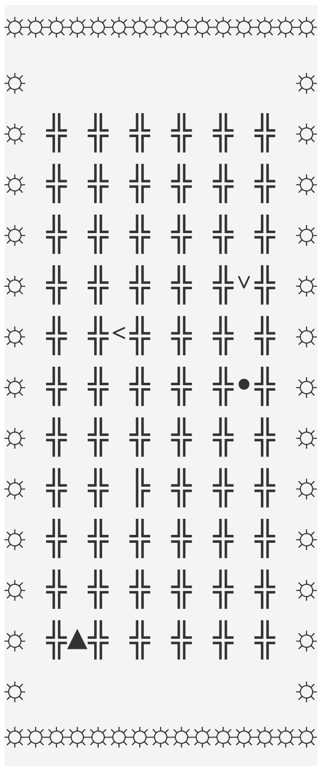
Here your@email.com is the email you specified when registering on the server, a code is your security token, you can get it from the address bar of the browser after registration / login

After connecting, the client will regularly (every second) receive a string of characters - with the coded state of the field. The format is **^ board = (. *) $**

With this regexp, you can bite the string board. Here is an example of a line from the server:

board=
☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼   ☼☼╬╬╬╬╬╬☼☼╬╬╬╬╬╬☼☼╬╬╬╬╬╬☼☼╬╬╬╬╬ᵛ╬☼☼╬╬<╬╬╬╬☼☼╬╬╬╬╬•╬☼☼╬╬╬╬╬╬☼☼╬╬╠╬╬╬☼☼╬╬╬╬╬╬☼☼╬╬╬╬╬╬☼☼╬▲╬╬╬╬╬☼☼
☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼

Length of the line is equal to the area of the field. If you insert a newline character every sqrt (length (string)) of characters, you get a readable image of the field.



The first character of the line corresponds to the cell located in the lower left corner and has the coordinate [0, 0]. In this example, the position of the tank (symbol ▲) is [3, 2], and the projectile (symbol •) is [11, 7].

Message decoding

Empty place - on which the tank can move:
GROUND (‘ ‘)

Indestructible wall:
WALL (‘☼')

Destroyed player's tank (a new one will appear in the next second):
DEAD (‘Ѡ')

This is followed by images of destructible walls. The numbers here indicate the number of shots required to destroy the walls completely. *Walls are restored over time*.
CONSTRUCTION ('╬', 3)
CONSTRUCTION_DESTROYED_DOWN ('╩', 2)
CONSTRUCTION_DESTROYED_UP ('╦', 2)
CONSTRUCTION_DESTROYED_LEFT ('╠', 2)
CONSTRUCTION_DESTROYED_RIGHT ('╣', 2),
CONSTRUCTION_DESTROYED_DOWN_TWICE ('╨', 1)
CONSTRUCTION_DESTROYED_UP_TWICE ('╥', 1)
CONSTRUCTION_DESTROYED_LEFT_TWICE ('╞', 1)
CONSTRUCTION_DESTROYED_RIGHT_TWICE ('╡', 1)
CONSTRUCTION_DESTROYED_LEFT_RIGHT ('│', 1)
CONSTRUCTION_DESTROYED_UP_DOWN ('─', 1)
CONSTRUCTION_DESTROYED_UP_LEFT ('┌', 1)
CONSTRUCTION_DESTROYED_RIGHT_UP ('┐', 1)
CONSTRUCTION_DESTROYED_DOWN_LEFT ('└', 1)
CONSTRUCTION_DESTROYED_DOWN_RIGHT ('┘', 1)

This projectile, both friendly and enemy:
BULLET (‘•')

Player tank:
TANK_UP ('▲')
TANK_RIGHT ('►')
TANK_DOWN ('▼')
TANK_LEFT (‘◄')

Enemy tank (autobot or opponent player)
OTHER_TANK_UP ('˄')
OTHER_TANK_RIGHT ('˃')
OTHER_TANK_DOWN ('˅')
OTHER_TANK_LEFT ('˂')
The game is step-by-step, every second the server sends to your client (bot) the status of the updated field at the current time and waits for the command to respond to the tank. For the next second, the player must have time to give command to the tank. If you do not have time - the tank remains in place.

There are several commands: **UP**, **DOWN**, **LEFT**, **RIGHT** - lead to the rotation and movement of the tank in a given direction to 1 cell; **ACT** - projectile shot. Motion commands can be combined with shot teams, separated by commas - this means that in one tact of the game there will be a shot and then movement (**LEFT**, **ACT**) or vice versa (**ACT**, **LEFT**)

The first task is to write a websocket client that connects to the server. Then get the tank to obey the command. The second task is to play a meaningful game and win.

For ANN I've taken the feedforward neural network, with three layers:

The first layer has 94 perceptrons

I've decided to measure 94 feature from the board's data
My features contain:
Information about 9x9 cells around the tank
The number of cells to the wall UP, DOWN, LEFT, RIGHT
Coordinates (X and Y) about 5 closest enemies. Distance is computed using Manhattan's distance

The second layer has 25 perceptrons

As a rule of thumb, I've decided to have 25 as a number of hidden layers in order to be in time to compute it during each movement

The third layer has 10 perceptrons

The last layer has 10 perceptrons because there're 10 different decisions can be made:

```
case 0 => Direction.LEFT.toString   + ',' + Direction.ACT.toString
case 1 => Direction.RIGHT.toString  + ',' + Direction.ACT.toString
case 2 => Direction.DOWN.toString   + ',' + Direction.ACT.toString
case 3 => Direction.UP.toString     + ',' + Direction.ACT.toString
case 4 => Direction.ACT.toString
case 5 => Direction.LEFT.toString
case 6 => Direction.RIGHT.toString
case 7 => Direction.DOWN.toString
case 8 => Direction.UP.toString
case 9 => “"
```

Conclusion

The computing world has a lot to gain fron neural networks. Their ability to learn by example makes them very flexible and powerful. Furthermore there is no need to devise an algorithm in order to perform a specific task; i.e. there is no need to understand the internal mechanisms of that task. They are also very well suited for real time systems because of their fast responseand computational times which are due to their parallel architecture.

Neural networks also contribute to other areas of research such as neurology and psychology. They are regularly used to model parts of living organisms and to investigate the internal mechanisms of the brain.

Perhaps the most exciting aspect of neural networks is the possibility that some day 'consious' networks might be produced. There is a number of scientists arguing that conciousness is a 'mechanical' property and that 'consious' neural networks are a realistic possibility.

Also, I would like to state that even though neural networks have a huge potential we will only get the best of them when they are intergrated with computing, AI, fuzzy logic and related subjects.

Finally, in my opinion, GAs are good to teach a NN but they will not be my first choice. Instead, I will try to look for better ways to learn the hyper-parameters of a NN. If there are any, that is.

However, if in the future I get access to a lot of processing power, I will be sure to try this method out.

Attached Code-base

This python script implements genetic-algorithms

It does next steps:

1) Implements genetic-algorithm (executes 10 servers with AI bots)

2) Executes it every 30 seconds. Then it retireves scores of each servers using REST API

3) Analyze score and use this data for genetic algorithms

```
#!/usr/bin/env python3
import requests
import random
import string
import argparse
import json
import re
import time
import subprocess
import os

NN_SIZE = 3285

def gen_word(k=10):
    return ''.join(random.choices(string.ascii_letters, k=k))

def remove(url):
    requests.get(url + '&remove=true')

def remove_bots(urls):
    for url in urls.values():
        remove(url)

def get_scores(url):
    # http://157.230.127.144:8080/codenjoy-contest/rest/player/
53521m6i2watnfa09nbi/73866122510/wantsToPlay/battlecity
    new = url.replace('board', 'rest').replace('?code=', '/') + '/
wantsToPlay/battlecity'
```

```
    data = json.loads(requests.get(new).text)
    return {p['name']: int(p['score']) for p in data['players']}

def extract_name(url):
    # http://157.230.127.144:8080/codenjoy-contest/board/player/
ixtuvgc22apnehn8sprn?code=3033159999295643265
    return re.search('player/(.*)\?', url).group(1)

def register(srv, game_name):
    email = gen_word() + '@' + gen_word() + '.com'
    readable_name = gen_word() + ' ' + gen_word()
    password = gen_word()
    url = 'http://{}/codenjoy-contest/register'.format(srv)

    data = {'data': '',
            'email': email,
            'readableName': readable_name,
            'password': password,
            'gameName': game_name}

    r = requests.post(url, data=data)
    return (extract_name(r.url), r.url)

def register_bots(srv, game_name='battlecity', cnt=20):
    urls = []
    for i in range(cnt):
        urls.append(register(srv, game_name))
    return dict(urls)

def run(sol, url, featureFile=''):
    # java -Dalgorithm=S -Durl=http://157.230.127.144:8080/
codenjoy-contest/board/player/oql9oatkpelq9mxii1ef?
code=7585365918951697611 -jar all.jar
    return subprocess.Popen(['/usr/bin/java', '-Dalgorithm=' +
sol, '-Durl=' + url, '-DfeatureFile=' + featureFile, '-jar',
'target/battlecity-engine-jar-with-dependencies.jar'],
stdout=subprocess.DEVNULL)
```

```
def cross(nn1, nn2):
    till = random.randrange(len(nn1))
    return [lerp(nn1[i], nn2[i]) for i in range(till)] +
[lerp(nn2[i], nn1[i]) for i in range(till, len(nn2))]

def lerp(x, y, c=0.3):
    return x + (y - x)*c

def gen_weight():
    return random.uniform(-1000.0, 1000.0)

def mutate(nn):
    new = []
    for i in range(len(nn)):
        if random.random() < 0.3:
            new.append(gen_weight())
        else:
            new.append(nn[i])
    return new

def gen_nn():
    return [gen_weight() for i in range(NN_SIZE)]

def print_nn(nn, filename):
    with open(filename, 'w') as f:
        f.write(' '.join(map(str, nn)))

def read_nn(filename):
    with open(filename, 'r') as f:
        line = f.readlines()[0]
    return list(map(float, line.split(' ')))

def rank_nns(url, nns, timeout=30):
    print('Registering bots...')
    urls = register_bots(args.url, cnt=len(nns))
    nns = dict(zip(urls.keys(), nns))
    print('Dumping NNs...')
    for name, nn in nns.items():
        print_nn(nn, 'nns/' + name)

    print('Starting...')
    ps = list(map(lambda i: run('N', i[1], 
featureFile='nns/'+i[0]), urls.items()))

    print('Waiting {} seconds...'.format(timeout))
    time.sleep(timeout)
    for p in ps:
        p.terminate()

    scores = get_scores(list(urls.values())[0])
    print(sorted(scores.values(), reverse=True))
    result = sorted(nns.items(), key=lambda p: scores[p[0]], 
reverse=True)

    remove_bots(urls)
    print('Bots removed.')

    for name in nns.keys():
        os.remove('nns/' + name)
    print('NN dumps cleaned.')

    return list(map(lambda p: p[1], result))

parser = argparse.ArgumentParser()
parser.add_argument('url', type=str)
args = parser.parse_args()

# nns = [gen_nn() for i in range(10)]
nns = [read_nn('top/' + str(i)) for i in range(10)]

for i in range(1000):
    print('Epoch ' + str(i))
    nns = rank_nns(args.url, nns)
    nns = nns[:5] + [cross(nns[random.randrange(5)], 
nns[random.randrange(5)]) for i in range(5)]
    if random.random() < 0.1:
        j = random.randrange(10)
        nns[j] = mutate(nns[j])
    for i in range(10):
        print_nn(nns[i], 'top/' + str(i))
```

Server’s code

Scala code which parses specific server’s features from file

```
import scala.io.Source
object FileParser {
  val M1_W = 95
  val M1_H = 25
  val M2_W = 26
  val M2_H = 25
  val M3_W = 26
  val M3_H = 10

  def parseFile(filePath: String): (Matrix, Matrix, Matrix) = {
    val lines = Source.fromFile(filePath).getLines().toArray
    assert(lines.size == 1)
    val numbers = lines(0).split(" ").map(_.toDouble)
    assert(numbers.size == (M1_H * M1_W + M2_H * M2_W + M3_H *
M3_W))
    val (matrix1, rest1) = buildMatrix(numbers, M1_W * M1_H, M1_W)
    matrix1.assertHeightAndWidth(M1_H, M1_W)
    assert(rest1.size == (M2_H * M2_W + M3_H * M3_W))
    val (matrix2, rest2) = buildMatrix(rest1, M2_W * M2_H, M2_W)
    matrix2.assertHeightAndWidth(M2_H, M2_W)
    assert(rest2.size == (M3_H * M3_W))
    val (matrix3, rest3) = buildMatrix(rest2, M3_W * M3_H, M3_W)
    matrix3.assertHeightAndWidth(M3_H, M3_W)
    (matrix1, matrix2, matrix2)
  }

  private def buildMatrix(numbers: Array[Double], size: Int,
width: Int): (Matrix, Array[Double]) = {
    (Matrix(numbers.take(size).grouped(width).toArray),
numbers.drop(size))
  }
}

case class Matrix(value: Array[Array[Double]]) {
  def assertHeightAndWidth(height: Int, width: Int): Unit = {
    assert(value.size == height)
    value.foreach(row => assert(row.size == width))
  }
}
```

Java code which process board based on features

```
import com.codenjoy.dojo.battlecity.client.Board;
import java.util.stream.Collectors;
import java.util.Collections;
import java.util.List;
import java.util.ArrayList;
public class BoardProcessor {
    static int processBoard(double[][] matrix1, double[][]
matrix2, double[][] matrix3, Board board) {
        List<Double> features = board.extractFeatures();
        features.add(1.0);
        features = multiply(matrix1, features);
        features =
features.stream().map(Math::tanh).collect(Collectors.toList());
        features.add(1.0);
        features = multiply(matrix2, features);
        features =
features.stream().map(Math::tanh).collect(Collectors.toList());
        features.add(1.0);
        features = multiply(matrix3, features);
        features =
features.stream().map(Math::tanh).collect(Collectors.toList());
        double m = Collections.max(features);
        for (int i = 0; i < features.size(); i++) {
            if (Math.abs(features.get(i) - m) < 0.001) return i;
        }
        return -1;
    }
    static List<Double> multiply(double[][] m, List<Double> v) {
        List<Double> res = new ArrayList<>();
        for (int i = 0; i < m.length; i++) {
            res.add(multiply(m[i], v));
        }
        return res;
    }
    static double multiply(double[] v1, List<Double> v2) {
        double res = 0.0;
        for (int i = 0; i < v1.length; i++) {
            res += v1[i] * v2.get(i);
        }
        return res;
    }
}
```

Scala code which makes move decision based on board's processing

```
import com.codenjoy.dojo.battlecity.client.Board
import com.codenjoy.dojo.client.Solver
import com.codenjoy.dojo.services.Direction
```

```
class NNProcessor(matrix1: Matrix, matrix2: Matrix, matrix3:
Matrix) extends Solver[Board] {
  override def get(board: Board): String = {
    if(board.isGameOver) return ""
    mapResult2Position(
      BoardProcessor.processBoard(
        matrix1.value,
        matrix2.value,
        matrix3.value,
        board
      )
    )
  }
  def mapResult2Position(result: Int): String = {
    result match {
      case 0 => Direction.LEFT.toString   + ',' +
Direction.ACT.toString
      case 1 => Direction.RIGHT.toString  + ',' +
Direction.ACT.toString
      case 2 => Direction.DOWN.toString   + ',' +
Direction.ACT.toString
      case 3 => Direction.UP.toString     + ',' +
Direction.ACT.toString
      case 4 => Direction.ACT.toString
      case 5 => Direction.LEFT.toString
      case 6 => Direction.RIGHT.toString
      case 7 => Direction.DOWN.toString
      case 8 => Direction.UP.toString
      case 9 => ""
      case r => throw new RuntimeException(s"Incorrect max index
in last NN result, should be [0..9] but actually is $r")
    }
  }
}
object NNProcessor {
  def apply(filePath: String): NNProcessor = {
    val (matrix1, matrix2, matrix3) =
FileParser.parseFile(filePath)
    new NNProcessor(matrix1, matrix2, matrix3)
  }
}
```

References

1. An introduction to neural computing. Aleksander, I. and Morton, H. 2nd edition

2. Neural Networks at Pacific Northwest National Laboratory
   http://www.emsl.pnl.gov:2080/docs/cie/neural/neural.homepage.html

3. Industrial Applications of Neural Networks (research reports Esprit, I.F.Croall, J.P.Mason)

4. A Novel Approach to Modelling and Diagnosing the Cardiovascular System
   http://www.emsl.pnl.gov:2080/docs/cie/neural/papers2/keller.wcnn95.abs.html

5. Artificial Neural Networks in Medicine
   http://www.emsl.pnl.gov:2080/docs/cie/techbrief/NN.techbrief.ht

6. Neural Networks by Eric Davalo and Patrick Naim

7. Learning internal representations by error propagation by Rumelhart, Hinton and Williams (1986).

8. Klimasauskas, CC. (1989). The 1989 Neuro Computing Bibliography. Hammerstrom, D. (1986). A Connectionist/Neural Network Bibliography.

9. DARPA Neural Network Study (October, 1987-February, 1989). MIT Lincoln Lab. Neural Networks, Eric Davalo and Patrick Naim

10. Assimov, I (1984, 1950), Robot, Ballatine, New York.

11. Electronic Noses for Telemedicine
    http://www.emsl.pnl.gov:2080/docs/cie/neural/papers2/keller.ccc95.abs.html

12. Pattern Recognition of Pathology Images
    http://kopernik-eth.npac.syr.edu:1200/Task4/pattern.html

13. https://towardsdatascience.com/artificial-neural-networks-optimization-using-genetic-algorithm-with-python-1fe8ed17733e

14. https://blog.coast.ai/lets-evolve-a-neural-network-with-a-genetic-algorithm-code-included-8809bece164

15. https://towardsdatascience.com/gas-and-nns-6a41f1e8146d