Evolving Algorithms: The EvoSim Engine

Building fast and effective evolutionary algorithms

J.D. Bruce
Rev 1: Dec 2016
www.j-d-b.net

Abstract

This paper describes the EvoSim Engine, a general purpose evolutionary algorithm for evolving virtual agents composed of computational units which perform operations on dynamic memory arrays. The system provides a Turing complete search space while minimizing the effort required to find solutions in the space. The engine takes a lot of inspiration from natural biological systems but does not aim to realistically simulate those systems, it aims to understand them in terms of information theory and computer science and make use of the underlying principles to achieve fast rates of evolution.

Introduction

The EvoSim Engine is an open source application designed to simulate processes in biology, genetics, and nature in order to act as a training algorithm for so called "virtual creatures", which are referred to as agents within the code base. However the goal isn't to be as realistic as possible, the goal is to apply principles which produce the fastest possible rate of evolution. Unlike other training algorithms, genetic algorithms allow us to train essentially any model because we aren't using specific mathematical equations to reduce the error rate, we are essentially trying a bunch of random stuff and then hoping the process of evolution will refine the best solution while weeding out the worst solutions over time.

Since we have such a wide array of options, an important problem is finding which models work best with the process of evolution. Many programmers attempting to write their first evolution simulator / genetic algorithm choose to use well known models such as artificial neural networks (ANNs), and that does work if done properly. However a feed forward network is a very restrictive model, it can only solve problems in a very specific way. Many problems are best solved with algorithms that make use of dynamic memory arrays and a wide range of instructions that operate on the memory. ANNs with recurrent connections can have a form of memory but they are still fairly restricted and must develop complex networks to achieve simple memory tasks.

The Best Building Blocks

Why restrict ourselves to these networking ideas when we can evolve any system we want? When we are trying to simulate evolution, we should be paying attention to the environment we're attempting to evolve the creatures in. Unlike real living creatures, virtual creatures aren't evolving in a world with particle physics, they are evolving within the architecture of our computers, a world where everything is about CPU instructions reading and writing digital information to RAM. The original idea of a Universal Turing Machine is essentially the same thing, just with infinite tape/memory and a little reading head which moves along the tape reading and writing to it based on some rules. Our computers are the real world example of that theoretical machine.

Imagine if the only way you could write an algorithm was by connecting a bunch neurons together, it wouldn't be very easy to find solutions. Likewise, many programmers wouldn't use assembly to do everything, they'd often prefer to use a higher level language which makes everything a lot easier. So we could attempt to evolve neural networks, or virtual Turing machines, but we would prefer to make it as easy as possible for evolution to stumble upon a solution so there are probably better models we can try to evolve. The key problem here is really finding which building blocks work the best with the process of evolution, what type of building blocks provide flexibility but also simple and efficient solutions.

We would prefer something which has a Turing complete search space but such a search space is truly massive so we also want to make it easy for evolution to find solutions within the search space. Other common models include things like logic circuits, with only a NAND gate or a NOR gate it's possible to create any other logic gate so that approach is very flexible. However we have to put together quite a few logic gates just to create a logic circuit which can add two integers together so it may not be easy to find solutions using such low level building blocks. We don't want the system to spend ages learning how to put together logic gates in the correct combination just so it can add two numbers together.

At this point you may be wondering if it's a good idea to simply evolve algorithms using a high level programming language. One issue with this approach is that most languages have very strict syntax rules and it's easy to create invalid code. Even if the code does compile it could still suffer from run time errors. As a result nearly none of the code produced by the process of evolution would work and it would be very hard to find solutions. So another key point here is that we want to use building blocks which don't totally explode if we put them together in the wrong way. We want something that will output an answer regardless of how we put together the building blocks, and has the other properties described so far.

Modules and Nodes

The approach taken with the EvoSim Engine is to use building blocks which perform basic but useful operations such as arithmetic and comparison operations, etc. Each agent in the simulation has a collection of arrays which are operated on by these so called "nodes". Nodes are grouped into objects referred to as "modules". Each module has two “registers" which nodes may read and write to. These registers are also linked to the arrays in the agent collection via pointers. Each register has two pointers, one which it uses to read values into the register and one which it uses to write values from the register to an array. Some arrays such as the agent input array are read-only so the output pointers of a module will only point to arrays with write permission.

What a module does is controlled by the nodes it contains. Some nodes will instruct data to be read from or written to the registers, other nodes may alter a register pointer, or simply compute the sum of the two registers and save the value in one of the two registers, some nodes may even tell the module to go back to the first node or some other node, which can allow loops to form in the module algorithm. It should be easy to create any type of algorithm using the different node types but too many types could make the search space become overly large and difficult to work with. The reason for grouping the nodes into modules is primarily to encourage modularity and code reuse within the agents. This a crucial concept within both programming and genetics.

ANNs may not be suitable for all problems but our brain is clearly a very powerful machine when it comes to solving problems. Each neuron in the brain has a fairly similar structure but when you put them together in the right way they can do very interesting things. Likewise, each cell in a human body is fairly similar but creates a very complex being. In fact we all started as a single zygote cell which multiplied and eventually formed a human. So reusing a relatively simple design to form

complex systems is clearly a very powerful idea used by evolution. The EvoSim Engine applies this concept by using chromosomes which contain "blueprints" for modules so the same module can be constructed many times.

The process can be likened to the way proteins are built from a set of amino acids, and those amino acids are built from basic elements. The same protein will often be built many times based on instructions held in the DNA. The virtual chromosomes used by the EvoSim Engine are designed to encourage the reuse of modules. An agent computes an answer simply by running each module in a sequential fashion, although it should be possible to make the modules compute the answer in a parallel fashion with enough effort. The output of an agent is obtained simply by reading the values stored in the agents output array. An agent has several different arrays that the modules can read and/or write to, so far only the input and output arrays have been mentioned.

Dynamic Memory Arrays

There are 3 other arrays that every agent has; the working array, the community array, and the heritable array. The working array is simply an array that acts as a form of memory for the agent since the input array isn't writable and the output array must be used for outputting a result. The community array is an array shared between all the agents in the same "species". This gives agents within the same species a way to communicate between themselves and allows for complex swarm behavior to arise among agents. This type of behavior could be further encouraged by having a fitness function which rewards cooperation and coherence between agents. The heritable array is much like the working array, but is inherited by offspring.

The reasoning behind such an array is to allow the genetic makeup of an agent to change during its life time and pass on those changes, which has been proven to occur with humans. For example certain viruses can alter our DNA and inject new sequences, exposure to certain chemicals can alter your DNA, and our immune system can shuffle around DNA sequences to produce novel antibodies. This means the "blind watch maker" doesn't need to be so blind, our genetic data can adapt and change in response to the environment. Science refers to this concept as transposable elements or "jumping genes" and it's how creatures can pass on adaptations acquired during life without waiting for evolution to stumble upon the solution.

Modules can read and write to the heritable array and what a module does can depend on the contents of the heritable array. Since the contents of the array gets passed onto offspring, it has the potential to act as a sort of operating system for the agent. This is some what akin to the concept of coding and non-coding DNA. The data held in the virtual chromosomes describes how to build the modules, but the heritable array is also a type of genetic data and it has the ability to control how the nodes behave, like software for the modules. Whether the process of evolution actually exploits this ability depends on chance and the type of problem you're trying to solve, but there's a high chance some modules will interact with the heritable array.

Evolving the agents is done by filling the input arrays with data, running the modules, then using the output array to determine the fitness. This process is repeated until reaching the end of the current training data set. Each iteration every agent will lose some health proportional to its fitness. Generally it's a good idea to use multiple training data sets and choose a random set every "generation". Having multiple training data sets and choosing them at random helps prevent over-fitting for a specific set of data so when you test the agents using a data set they weren't trained with they should still perform well. Other methods often used to prevent over-fitting include the dropout method, entropy injection, and the McNemar test.

Generalization and Diversity

The dropout method simply applies a random chance for a node to stop working. This encourages general solutions which can handle a bit of unexpected behavior. Entropy injection is a very similar method, the input and output values of nodes are injected with a bit of random noise. The EvoSim Engine emulates access to an entropy pool by having nodes which just return random numbers using a PRNG. The McNemar test is a bit more complex but it's a very robust statistical method which can compare agents and say with confidence which agent is the best based on how often it was right when others were wrong, and vice versa. Depending on the problem it could be helpful in determining the fitness value for each generation.

The fitness value will affect how quickly the health of an agent will decrease and once the health of an agent reaches 0 it will die. Every iteration new agents are created to replace the dead agents. The process works a bit like a lottery, to decide who gets to have a child each agent gets assigned a random number and the two agents with the highest numbers will reproduce. However it's a little more complex than that because we must choose two agents from the same species, so first we pick a genus, then a species, then agents. The EvoSim Engine uses the term "genus" to mean a set of species which are related but have differences like slightly more or less chromosomes, a bit like the difference between a dog and wolf.

To give better agents a higher chance of having offspring we adjust the random range so that an agent with a higher fitness has a larger range of possible numbers so it has a higher chance of getting a large number. Each genus and each species also have fitness values which can be used the same way to determine which genus and which species has the highest chance of winning. This is a nice approach to creating offspring because it avoids complex probability tables. Splitting the agents into separate species of a certain genus is very useful because it encourages competition and diversity through speciation. Training a single species for many generations wont necessarily evolve well if they are just terrible designs to start with.

If all the agents in a species die then that species will go extinct and if all the species in a genus go extinct then the genus will go extinct. This forces each species to fight for space since the population size is limited. The worst species will tend to die off very quickly, leaving the remaining species to fight for dominance. This will typically continue until a small fraction of the species are remaining but they now perform much better than the first generation. This is good way to find agents which may be worth evolving further. You save the genetic data of the best agents and then import them and use them to create a new population which is composed of all the most promising agents then pit them against each other.

The Virtual Chromosomes

Each agent actually has two sets of chromosomes because one set of chromosomes is inherited from each parent like humans. Both copies of each chromosome are similar but can have some differences which make them produce modules which behave differently. The logic behind this is to have something similar to "alleles" which are variations of the same basic "genes" floating among the gene pool of a species. Since the modules created by a pair of chromosomes can behave differently we need a method of deciding which module to run or if we should run both. This leads to a system where modules can be "dominant" and "recessive", analogous to the way genes can be dominant and recessive, or even co-dominant.

To decide which version of the module will run, we give each module a level of "impact" which is stored in the chromosome. If the impact of the module from the mother chromosome is larger than the impact of the fathers equivalent module, then the module from the mother will run and the one from the father will not, and vice versa. Since the system is Turing complete we have no idea how long a module could run for, so we also assign a limit to the number of operations an agent can perform before it must stop. If the impact of both modules are equal then both modules will run but the second module may not get to run if the first module reaches the computation limit. This behavior can easily be modified to experiment with different ideas.

Since humans have two copies of each gene, one can often dominate the other. However in many cases it is also possible for both alleles to be expressed, called biallelic expression. In reality it's much more complicated than that but the goal isn't really to simulate exact biological systems, we generally prefer efficient approximations and rough emulations that achieve the same result as the real thing. However a module could also be "recessive" simply because it doesn't do much and gets overshadowed by the activity of its partner module, which is much closer the way recessive and dominant traits actually work in reality. This type of system is theoretically useful because it encourages successful dominant traits.

If a dominant trait is not working it should be weeded out by survival of the fittest since agents with those traits will quickly die. Over time the prevalence of modules which do something useful should increase and their impact value should grow as different variants of the module compete for survival. There is a reason these types of systems work well for evolution in the real world, and even if we are using what seem to be rather crude approximations of complex biological systems, they often provide great benefits because the underlying principles don't care whether a system is made of real particles or digital bits.

Agent Reproduction

Reproduction works simply by taking two agents from the same species, then generating a "gamete" from each agent and using those gametes to produce the final "zygote", and applying some mutations to the final genome. That genome can then be used to construct a new agent based on the data contained in the genome. Since each parent should only provide one copy of each chromosome, they must produce something like an egg or sperm cell (a gamete). The gamete contains a single set of chromosomes so you need two gametes to form a complete zygote which has two copies of each gene. This is essentially how it works with humans and how it works in the EvoSim Engine; agents produce gametes which can be combined to form a zygote.

Cells have a rather clever way of generating gametes from a complete set of chromosomes called meiosis. Each chromosome in the gamete has an equal chance of being from the mother or father, a single chromosome may even be a mix of both mother and father chromosomes sliced together at different points. The EvoSim Engine essentially replicates this behavior when generating a gamete from an existing agent, it chooses chromosomes at random and may also combine parts of both chromosomes, allowing chromosomes to change over time. So with two agents from the same species, or even just one agent, two gametes can be produced and used to form the complete genome of a new creature which can then be constructed.

The benefit of having this type of chromosomal mechanism is that the "DNA" or genetic material gets placed into modular packets which can persist through a population and fill different purposes. This should theoretically work better than the usual approach of treating DNA as a simple bit string and mixing the strings together in a simple fashion. Having two copies of each chromosome in each agent is also theoretically beneficial because it encourages variation among a species and allows for the production of diverse gametes which contain different combinations of chromosomes without

harming their purpose. During the reproduction process the heritable array is also passed onto the offspring as mentioned previously.

In reality the process of reproduction is far more complicated than anything the EvoSim Engine does. Living creatures often have complex incubation systems which play a role in determining how a creature is constructed and the reproduction mechanism its self is able to evolve over time, a very interesting but also very complex idea which is difficult to simulate in an effective manner. The reproduction mechanism in the EvoSim Engine doesn't have that level of realism, it will not evolve and it requires that agents must be from the same species in order to reproduce, but it is possible for agents from different species to reproduce if their chromosomes are similar enough.

Procedural Generation

The human genome is a very messy thing and trying to understand it is like trying to understand the worlds most complex yet poorly written program. Evolution doesn't care if it creates messy and obscure code, as long as that code achieves the desired goal. A single gene wont necessarily be in one spot, it can be spread all through the chromosome, with so called non-coding or "junk DNA" in between. Things like proteins are built in a modular fashion by combining different amino acids to create a huge amount of variety and much more complexity than you would guess just by looking at the size of the genome, much like a procedural game can be small in size but contain a huge amount of procedurally generated content.

The EvoSim engine makes use of procedural generation by using the chromosome to store seed values. Those seed values are then used to deterministically generate a bunch numbers which are used to build modules. The numbers generated from the seed value will determine the number and type of nodes contained in the module, so how a module behaves will depend on the seed value used to generate it. The same module can be constructed many times by using the same seed value to create the module, which goes back to the idea of reusing components. The other nice benefit of this approach is a reasonably smaller genome size since we only need to store seed values for modules and not all the data required to construct the nodes.

However there is a possible downside to generating modules in this fashion. Even a small change to the seed value will have a large impact on the resulting module, which usually isn't a good thing, so it's worth examining how nature minimizes the impact of critical genetic errors. One of the main tactics is gene duplication, some sequences are replicated thousands of times throughout the human genome. What this does, among other things, is provide redundancy, meaning if one gene gets messed up too bad then those backup copies can be used. Another thing gene duplication achieves is the ability to modify a gene and adapt it to do something different than the original gene, but without messing around with the original gene.

This is another reason why having two copies of each chromosome and having dominant and recessive modules is very important, it reduces the risk of critical errors and allows for innovation to occur without creating too many problems. Data in each chromosome specifies how many modules to create from a single seed, but there is also no reason the same seed value cannot occur more than once in a single chromosome. That provides something similar to gene duplication, even if one of the seeds changes, other copies will remain the same, so that specific module "blueprint" will not disappear from the chromosome just because of a small mutation, making it easier for the system to create variants without erasing the original.

Conclusion

When attempting to simulate the process of evolution, natural selection, survival of the fittest, there are many different factors which need to be taken into consideration. Biology and genetics is an extremely complex subject and the way real living creatures evolve is far more complicated than most people realize. Simplistic simulations of evolution fail to capture much of this complexity and as a result they often fail to produce an appreciable rate of improvement in the virtual creatures and they never produce the type of elaborate designs which can seem almost intelligently designed. By capturing many of the less obvious but highly important principles of biology and genetics we're able to get much better results.

However it's also important not to over-engineer, our computers have limited resources and evolution takes a lot of time to work properly so speed is very important. Evolution is clearly a very powerful process when it works correctly, so the focus should be on understanding the underlying core concepts so they can be approximated in an algorithmic fashion. The EvoSim Engine takes this approach with the goal being to evolve general algorithms consisting of modules which operate on dynamic memory arrays. This paper has provided a brief overview of how the EvoSim Engine works and the reasoning behind why it works that way. The full source code for the EvoSim Engine can be found on GitHub, the link to the repository can be found below.

Relevant Resources

https://github.com/JacobBruce/EvoSim-Engine

https://en.wikipedia.org/wiki/Evolutionary_algorithm

https://en.wikipedia.org/wiki/Genetic_algorithm

https://en.wikipedia.org/wiki/Recurrent_neural_network

https://en.wikipedia.org/wiki/Universal_Turing_machine

https://en.wikipedia.org/wiki/Assembly_language

https://en.wikipedia.org/wiki/Logic_gate

https://en.wikipedia.org/wiki/Swarm_intelligence

https://en.wikipedia.org/wiki/Transposable_element

https://en.wikipedia.org/wiki/Immune_system

https://en.wikipedia.org/wiki/Chromosome

https://en.wikipedia.org/wiki/Allele

https://en.wikipedia.org/wiki/Gamete

https://en.wikipedia.org/wiki/Zygote

https://en.wikipedia.org/wiki/Meiosis

https://en.wikipedia.org/wiki/Dominance_(genetics)

https://en.wikipedia.org/wiki/Dropout_(neural_networks)

https://en.wikipedia.org/wiki/Entropy_(information_theory)

https://en.wikipedia.org/wiki/McNemar's_test

https://en.wikipedia.org/wiki/Noncoding_DNA