Neuroevolution - Neat

Points	Submission due	Status
0 / 25	Feb 18, 2025 00:01	No graded result

Tasks:

Neuroevolution

Neuroevolution uses evolutionary algorithms to train neural networks and is an alternative to the more traditional approach of using gradient-based optimisation. In Neuroevolution, a population of neural networks is initialised with random attributes, and each individual is evaluated on the given task. Individuals that perform well are selected to produce offspring with modified attributes using mutation or crossover operations. The process of evaluation and reproduction continues iteratively over many generations, allowing individuals to incrementally improve their performance on a given task.

Algorithm Description

In the final assignment, you have to implement the *Neuroevolution of Augmenting Topologies* (*NEAT*) algorithm. This algorithm evolves the topology and weights of neural networks simultaneously, which frees the developer from the tedious task of finding an optimal network structure through trial and error.

Neat **represents genes** as a combination of two sub-gene lists: *neuron* and *connection* genes. The former gene type models the neurons of a neural network by defining properties such as their activation functions or position in the neural network (input, output, hidden neurons). On the other hand, connection genes represent the links between these neurons by specifying the source and target neurons, as well as the connection weight. Another crucial property of connection genes is their innovation number, which is assigned based on the historical origin of a gene and serves as a unique identifier. Whenever a new connection is added to a network, we have to check whether the same link in terms of connected source and target neurons has already been observed before within the evolution process. If this is the case, we assign the same innovation number to the freshly generated connection. Otherwise, we generate a new innovation number for the created connection and remember the assigned innovation number for future connection genes.

Neat's **mutation** operators can be divided into two groups: structural and non-structural. The former modifies a network's topological structure by adding a new hidden neuron or connection, while the latter operates on the attributes of existing connection genes. In this assignment, we will focus on two structural and two non-structural mutations:

- **Structural - Add Neuron:** New neurons are introduced by randomly disabling a connection in our network and inserting the neuron between the disabled connection by generating two new connections that lead in and out of the neuron. In order to beware the network of further changes, the connection leading into the new node obtains a weight value of 1, while the connection leading out of the neuron inherits the same weight as the disabled connection.
- **Structural - Add Connection:** New connections are introduced to the network by generating a single novel link with a randomised weight value between two previously disconnected neurons. For this assignment, we will restrict ourselves to feed-forward networks. Hence, new connections must not point backwards from a higher-level layer to a lower-level layer.
- **Non-Structural - Mutate Weights:** Weights are changed by adding random Gaussian noise to all connection weights of a given network.
- **Non-Structural - Toggle Connection:** Toggles the enabled status of a random connection in the network. If a connection gene is disabled, no information can flow from the source neuron of the connection to the target neuron.

Since structural mutations are NEAT's only measure to change a network's topology, we have to pay attention to the assignment of innovation numbers whenever we apply this kind of mutation.

The **crossover operator** randomly selects two parents and aligns their connection genes based on the assigned innovation numbers. Then, both connection gene lists are sequentially traversed, and specific rules are applied to each gene pair. Matching genes are inherited randomly from one of the two parents, while genes with differing innovation numbers are always inherited from the fitter parent. The three most important contributions of NEAT in the domain of neuroevolution are the introduction of innovation numbers, speciation, and the minimisation of the search space through incremental evolution. As stated above, innovation numbers can be considered identifiers for connection genes that enable the comparison of two connection genes, which is necessary to apply crossover effectively and realise the speciation of network structures.

Another crucial aspect of *Neat* is the use of **speciation** to protect innovative networks. This is necessary since innovative topologies introduced through structural mutations tend to initially decrease the fitness of a network, leading to architectural innovations having a hard time surviving even a single generation. Speciation ensures that neural networks with similar structures have to primarily compete against other networks in the same species, which gives them time to optimise their weights. The similarity of networks is calculated using the so-called *compatibility distance* δ, which is a linear combination of the number of *matching*, *disjoint* and *excess genes*:

$$\delta = \frac{c_1 D}{N} + \frac{c_2 E}{N} + c_3 \overline{W}$$

with the average weight difference of matching genes $\overline{W}$, the number of disjoint and excess genes D and E, the scaling coefficients c_1, c_2, c_3, and the normalizing factor N that represents the number of genes in the larger of both parents. Gene types are determined based on a comparison of the given innovation numbers:

- **Matching Genes** have the same innovation number.
- **Disjoint Genes** have mismatching innovation numbers within the range of both networks' innovation numbers
- **Excess Genes** have mismatching innovation numbers outside the range of both networks' innovation numbers.

Based on the computed compatibility distance δ, networks are considered similar if δ is below the *compatibility threshold* δ_t. The threshold is adjusted in each generation of the algorithm based on the number of species present in the current generation: if there are fewer species than desired, the threshold is reduced (e.g., by 0.3), and vice versa for too many species. Finally, to ensure that no species is able to take over the whole population, the fitness f_i of every member i of a species s is divided by the size of the species:

$$f_i' = \frac{f_i}{size(s)}$$

Thus, whenever a species grows too big, e.g. due to a single member of the species performing exceptionally well, fitness sharing reduces the number of children the given species is allowed to produce, creating more room for other species.

By optimising the topology and weights of networks simultaneously, *Neat* has to cope with a much bigger search space than other learning algorithms that only focus on the weights of a network. *Neat* minimises its search space by generating an **initial population** consisting of minimised network structures. These initial networks contain no hidden neurons and have a single connection from every input to each output neuron. Innovation in the form of new network structures is added via mutation and can only survive over several generations if it contributes positively to solving the given problem.

Problem Environments

Your implementation will be evaluated in two maximisation optimisation tasks: simulating an *XOR* gate and solving the *Single-Pole Balancing* problem. Both problem environments expect as input from the network a double in the range of $[-1, 1]$.

In the first task, the goal is to represent *XOR* computations through a neural network by returning the following outputs given the respective inputs:

- (0,0) -> 0
- (1,0) -> 1
- (0,1) -> 1
- (1,1) -> 0

This may sound easy, but it is a non-trivial task for neural networks because they can only solve this problem if they contain at least one hidden neuron. Therefore, *Neat* must be able to introduce new neurons into the network and optimise the connection weights effectively. The fitness of a network is given by comparing its uninterpreted outputs for each of the four inputs against the expected outputs of the *XOR* gate. A network that simulates the gate correctly has a fitness of 16, while the worst possible fitness is 0.

The *Single-Pole Balancing* problem is a popular benchmark task in the Reinforcement Learning community. It involves a pole mounted on a cart that can be moved to the left and right on a horizontal track. Based on the position p of the cart, the velocity $\dot{p}$ of the cart, the angle of the pole θ, and the angular velocity of the pole $\dot{\theta}$, the agent has to decide in each time step in which direction the cart should be moved. After an

About Feedback Releases Privacy Imprint

agent has finished a balancing attempt, it is rewarded with a fitness equal to the number of time steps it managed to survive. The goal is to move the cart to the left or right so that the pole remains balanced without the cart leaving the track's boundaries.

Your Task

In this assignment, your task is to implement the *Neat* algorithm to solve the *XOR* and *Single-Pole Balancing* problem within an apporpriate number of evolutionary generations. You are free to add new fields, methods and constructors to all given classes and to create new classes within the given package directories. Please ensure to not modify existing method and constructor signatures and refrain from creating new package directories. Wherever you have to flesh out the implementation of existing methods, you'll find an `UnsupportedOperationException("Implement me!")`.

NetworkChromosome Encoding

First, we have to implement our `NetworkChromosome`s consisting of `ConnectionGene`s and `NeuronGene`s as governed by the `Neat` algorithm. To this end, you have to finalise the implementation of the `NeuronGene` and `ConnectionGene` classes, which are responsible for modelling the neurons and connections of a neural network. Next, we can define the `NetworkChromosome`, which assembles a list of `NeuronGene`s and `ConnectionGene`s into a fully functional neural network phenotype. The provided constructor expects a list of `ConnectionGene`s and a layer map that maps a `double` representing the layer depth in the range of $[0, 1]$ to a list of `NeuronGene`s placed at the respective layer depth. Within the layer map, we define the value of 0 to represent the input layer and a value of 1 to represent the output layer. Thus, hidden layers must be assigned a value between $(0, 1)$, such that a neuron placed at layer i with depth $i < j$ is closer to the input layer than a neuron placed at layer j. The `getOutput(List<Double> state)` method of the `NetworkChromosome` activates the neural network with an input representing the current state of a given task environment and outputs a list of network output values that should equal the number of output neurons present in the respective network.

In order to crate the initial population of neural networks in the *Neat* algorithm, you have to implement the `generate` method of the `NetworkGenerator`. Remember that *Neat's* initial population consists of neural networks with minimal network structures, i.e. fully-connected feed forward networks without hidden neurons. The number of required input and output neurons for each task can be queried via the `stateSize` and `actionInputSize` methods of the `Environment` interface. **Bias neurons will be realised by a single node that is placed at the end of the input layer list. This bias neuron has a constant activation value of 1 and is initially connected to every output neuron**. Since the network generator creates new connection genes, make sure to update the set of `Innovations` appropriately during the network generation in order to assign appropriate innovation numbers to the created connections.

Mutation & Crossover

To evolve the population of networks, you have to implement the mutation and crossover operator of *Neat*.

In `NeatMutation`, implement the `apply(NetworkChromosome parent)` method, which mutates the given parent by applying at least one of the four mutation strategies outlined above. Since structural mutations have a big impact on the resulting phenotype, *Neat* refrains from applying further non-structural mutations to the resulting mutant if a structural mutation has already been applied to it. Whenever you apply structural mutations, ensure to update the given list of `Innovations` appropriately.

For implementing the crossover operation, implement the `apply(NetworkChromosome parent1, NetworkChromosome parent2)` method in `NeatCrossover`. As mentioned above, the crossover operator aligns the connection genes of both parents and determines for each connection whether it is present in both parents or only in one of the parents. Genes that are present in both parents are inhereted randomly while genes present in only one parent are inherited only from the more fit parent.

Neat

Once we have our chromosome encoding and the evolutionary operators in place, we can start with realising our search algorithm *Neat* by implementing the `Neuroevolution` algorithm. *Neat* starts with creating an initial population of networks with minimised network structure and evaluates these networks on the given problem task. Next, all chromosomes are assigned to a species by iterating over all currently existing species and computing the compatibility distance δ between the given network and (a randomly chosen) representative of the species. If the compatibility distance δ falls below the threshold δ_t, the network is assigned to the respective species. In case the novel chromosome is not compatible to any species, a new species is created with the novel chromosome as its founding member.

Once all members of the population were assigned to a species, the algorithm continues by assigning each species the number of offspring it is allowed to produce based on the relative performance of the species. Keep in mind that in order to give species that host novel innovations some time to optimise their weights to the new network structure, we have to compute the performance of a species based on the shared fitness value of its members. Having assigned each species the number of offspring it is allowed to produce, we subset of the best-performing networks in a species. Finally, the cycle continues with replacing the previous population with the newly generated offspring, evaluating these offspring in the problem domain and assigning each offspring to a compatible species.

After implementing the *Neat* algorithm, please ensure to flesh out the `initialiseNeat(int populationSize, int maxGenerations)` method in the `Main` class by instantiating your *Neat* algorithm with the specified population size and maximum number of allowed generations. Please ensure that each *Neat* instance starts with a fresh state and that no information is shared between instances.

Functional Tests

1. **Correct Implementation of the Genotype** No results
2. **Correct Implementation of the Network Generator** No results
3. **Correct Implementation of the Mutation Operator** No results
4. **Correct Implementation of the Crossover Operator** No results
5. **Accepted Neat Results** No results

Coverage Tests (20% of points)

1. **Line Coverage >= 70%** No results
2. **Branch Coverage >= 60%** No results
3. **Mutation Score >= 55%** No results

Tips and Remarks

- You can execute the application by first building the application via `mvn package` and then executing `java -jar target/Neuroevolution-Neat.jar`. For instance, you can solve the *Single-Pole Balancing* with randomised initial states with *Neat* using a population size of 50 networks and a maximum of 50 generations over 30 repetitions via `java -jar target/Neuroevolution-Neat.jar -t CART_RANDOM -p 50 -g 50 -r 30`. Moreover, you can set the `-v` flag to visualise how the first agent evolved in the specified number of repetitions behaves in the *Single-Pole Balancing* task.
- You can visualise how an evolved agent behaves in the *Single-Pole Balancing* environment via the `Environment.visualise(Agent agent, CountDownLatch latch)` method. This might be helpful during debugging.
- One of the biggest challenges of implementing *Neat* is to assign neurons appropriate IDs to facilitate the assignment of innovation numbers to connection genes. We recommend using the same ID for neurons across networks but to restrict neuron IDs to be different within a single network. For instance, if a hidden neuron is placed between the second input node and the first output node, this neuron should always have the same ID, regardless of in which network it exists.
- The tests for the *Neat* algorithm will use a population size of 50 networks for the *XOR* and the *Single-Pole Balancing* without randomisation task. For the *Single-Pole Balancing* with randomisation task, a population size of 100 networks will be used. *Neat* will be executed 30 times for each task and is allowed to run for a maximum of 100 generations.
- If you find the need to generate random numbers, please do not create your own instances of `java.util.Random`. For the `-s` flag to work properly, use the `Randomness` class instead.
- Make sure to add the suffix `Test` to all your unit tests and that you place them in the appropriate `test` directory.
- If you make use of LLM tools such as ChatGPT, upload the prompts you send to the LLM together with the answers you obtained in a folder called `LLM`. Furthermore, ensure to annotate every piece of code you write with the help of LLM tools.
- The execution of all tests will take some time. Thus, we recommend **testing your implementation locally on your machine before pushing it to Artemis**.
    - You do *not* have to push every commit separately to Artemis.
    - If Artemis reports, "No corresponding result available", please wait a couple of minutes (~30 minutes) for the results to appear.
    - In case there are many submissions (e.g. shortly before the submission deadline), it may take longer for the results to appear since your build has to wait in a queue. You can check the current queue length at https://artemis.fim.uni-passau.de/ci-queue-length.html.
- **The Artemis pipeline has a timeout of 10 minutes for each stage (mutation testing, executing your unit tests, executing our functional tests). If the pipeline is interrupted due to the timeout, Artemis will report only partial results for the interrupted stage. ('Test was not executed')**
    - To rate-limit trial-and-error debugging via Artemis, we only run the tests for your latest push. I.e., if you are pushing a second time before you have received the results for the previous push, the tests will only continue to run for the second push and the not-yet-finished first test execution will be aborted.

Exercise details

Release date	Jan 28, 2025 14:00
Submission due	Feb 18, 2025 00:01
Complaint possible	No