Chapter 8. ES-HyperNEAT and Retina Problem

In Chapter8, you will learn about the ES-HyperNEAT extension of the HyperNEAT method, which we discussed in the previous chapter. As you learned in the previous chapter, the HyperNEAT method allows encoding of the larger-scale ANN topologies, which is essential to work in the areas where input data has a large number of dimensions, such as computer vision. However, despite all its power, the HyperNEAT method has a significant drawback - the configuration of the ANN substrate should be designed beforehand by a human architect. The ES-HyperNEAT method was invented to address this issue by introducing the concept of Evolvable Substrate. Through this chapter, you will gain hands-on experience with applying the ES-HyperNEAT method to solve the modular retina problem.

In this chapter, we cover the following topics:
- The concept of the evolvable substrate for the HyperNEAT algorithm and how it can be used to produce the appropriate configuration of the substrate during the evolution automatically.
- The basis of the modular retina problem and how evolving the modular structures within the substrate can solve it, by translating into the modular topology of the detector ANN.
- You will learn how to choose a suitable initial substrate configuration that helps the evolutionary process to discover the modular structures.
- We will discuss the source code of the modular retina problem solver along with the test environment, which can be used to evaluate the fitness of each detector ANN.

Technical Requirements

The following technical requirements should be met to execute the experiments described in this chapter:
- MS Windows 8/10, MacOS 10.13 or newer, Modern Linux
- Anaconda Distribution v. 2019.03 or newer

Evolvable Substrate HyperNEAT

The HyperNEAT method, which we discussed in the previous chapter, allows using the neuroevolution methods for a broad class of problems, which require the use of large-scale ANN structures to find a solution. This class of the problems spreads across multiple practical domains, including visual pattern recognition. The main distinguishing feature of all these problems is a high dimensionality of the input/output data.

In the previous chapter, you learned how to define the configuration of the substrate of the discriminator ANN to solve the visual discrimination task. You also learned that it is crucial to use appropriate substrate configuration that is aligned with the geometric features of the search space of the target problem. With the HyperNEAT method, you, as an architect, need to define the substrate configuration beforehand, using only your understanding of spatial geometry of the problem. However, it is not always possible to learn about all geometric regularities hidden behind a specific problem space.

If you design the substrate manually, you create an unintentional constraint on the pattern of weights drawn over it by CPPN. By placing nodes at specific locations in the substrate, you interfere with the ability of the CPPN to discover geometric regularities of the natural world. The CPPN should produce the connectivity pattern, which is perfectly aligned with the structure of the substrate that you provided, and connections are possible only between nodes of this structure.

Such limitation leads to unnecessary approximation errors, which defiles the outcomes when you use trained CPPN to create the topology of the solution-solver ANN (phenotype).

However, why should such limitations that are introduced with manual substrate configuration be inflicted in the first place? Would it be better if CPPN could elaborate on the connectivity patterns between nodes of the substrate that is automatically positioned in the substrate in the right locations? It seems that evolving connectivity patterns in the substrate provide valuable implicit hints that allow estimating nodes positions for the next epoch of the evolution. The method of substrate configuration evolution during the CPPN training got a name: Evolvable Substrate.

However, why should such limitations that are introduced with manual substrate configuration be inflicted in the first place? Would it be better if CPPN could elaborate on the connectivity patterns between nodes of the substrate that is automatically positioned in the substrate in the right locations? It seems that evolving connectivity patterns in the substrate provide valuable implicit hints that allow estimating nodes positions for the next epoch of the evolution. The method of substrate configuration evolution during the CPPN training got a name: Evolvable Substrate.

The implicit data allowing to estimate the position of the next node is the amount of the information encoded by the connectivity pattern in the specific substrate area. The areas with uniform distribution of the connection weights encode a small amount of information, thus requiring only a few substrate nodes in those areas. At the same time, substrate areas with large gradients of the connection weights are informationally-intensive and can benefit from additional nodes placed within those areas. When you place an additional node in such areas of the substrate, you allow CPPN to represent the much more granular encoding of the natural world. Thus, the placement of the nodes and the connectivity pattern can be mandated by the distribution of the connection weights while the CPPN produces the connection weights during the evolution.

The HyperNEAT represents each connection between two nodes of the substrate as a point in the four-dimensional hypercube. The Evolvable Substrate HyperNEAT algorithm extends the HyperNEAT by automatically placing fewer hyper-points in the areas of the hyper-cube with a lower variation of the connection weights. Thus, ES-HyperNEAT uses the information density as a primary guiding principle when determining the topology of the substrate during the evolution.

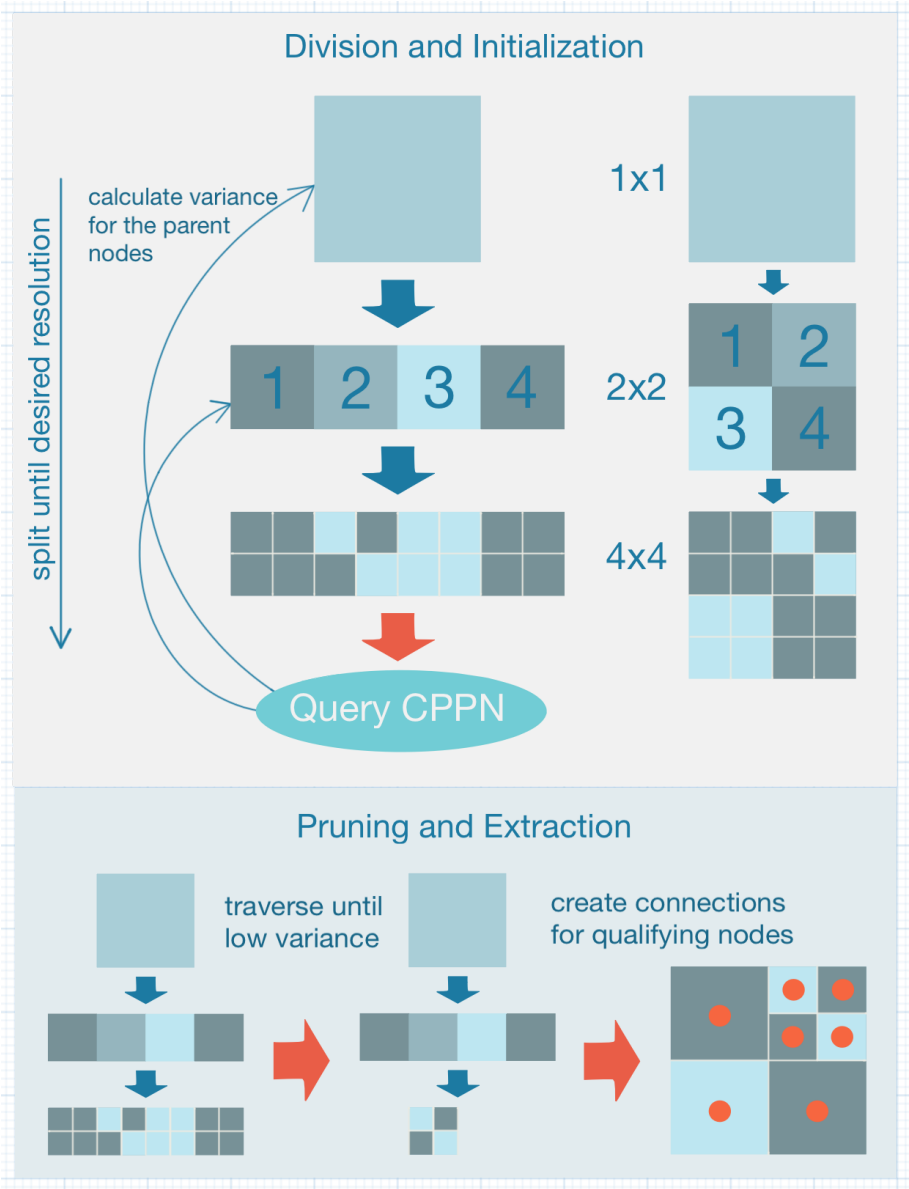
Quadtree for Information Extraction

For effective calculation of the information density within connectivity patterns of the substrate, we need to use an appropriate data structure. We need to employ the data structure, which allows an effective search through the two-dimensional substrate space at the different levels of granularity. In computer science, a long time ago was invented the data structure that perfectly fits the requirements described above. Such a structure is the quadtree.

The quadtree is a data structure allowing us to organize an effective search through the two-dimensional space by splitting any area of interest into the four sub-areas. Each of the subareas consequently becomes the leaf of a tree with the root node representing the initial region.

The ES-HyperNEAT employs a quadtree data structure to iteratively look for the new connections and nodes in the substrate, starting from the input and the output nodes predefined by the experimenter. Using quadtree to search for new connections and nodes is much more computationally effective than searching in the four-dimensional space of the hypercube.

The scheme of information extraction using the quadtree is shown in the following image:



The scheme of information extraction

The information extraction method depicted in the plot above has two major parts:

1. The division and initialization stage presented in the top part of the plot above. At this stage, the quadtree is created by recursively dividing the initial substrate area, which spans from (-1,-1) to (1,1). The division stops when the desired depth of the quadtree is reached. Now we have several subspaces that are fitted into substrate determining the initial substrate resolution (r). Next, for every node of the quadtree with a center at (x_i, y_i), we query the CPPN to find a connection weight (w) between this node and a specific input or output neuron at coordinates (a, b). When we have calculated the connection weights for the k leaf nodes in the subtree of the quadtree p, we are ready to calculate the information variance of the node p in the quadtree as following: $\sigma^2 = \sum_{i=1}^{k} (\bar{w} - w_i)^2$, where \bar{w} is the mean connection weight among k leaf nodes, and w_i is a connection weight to each leaf node.

   We can use this estimated variance value as a heuristic indicator of the information density in the specific subarea of the substrate. The higher this value, the higher the information density. The variance can be used to manage the information density in the specific subarea of the substrate by introducing the ***division threshold*** constant. If the variance is greater than the division threshold, then the division stage repeated until desired information density reached.

At this stage, we created an indicative structure that allows CPPN to decide where to make connections within the given substrate. The next stage of the processing places all necessary connections using a created quadtree structure.

2. The **pruning and extraction** stage represented in the bottom part of the plot above. This stage uses the populated quadtree structure from a previous stage to guarantee that more connections are expressed in the regions of higher variance. The quadtree traversed depth-first until the variance of the current node becomes smaller than a given ***variance threshold*** (\sigma_t^2) or until the current node has no children (i.e., have zero variance). For every quadtree node found by the depth-first search described above, we express the connection between the center of the node (x,y) and each parent node, which is already determined. The parent node can be determined either by an architect (input/output nodes) or become found at the previous runs of the information extraction method, i.e., from hidden nodes already created by the ES-HyperNEAT method. After this stage, the density of the nodes in the different regions of the substrate will correspond to the amount of information in that region.

In the following section, we discuss how to use the ES-HyperNEAT algorithm described above to find a solution for the modular retina problem.

> i: For more details about ES-HyperNEAT algorithm refer to the chapter ‘Overview of Neuroevolution Methods ’.

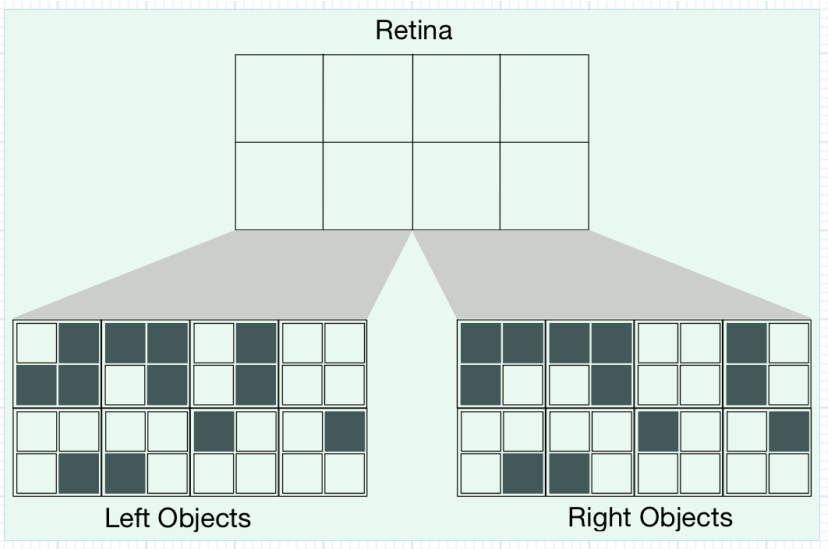
Modular Retina Problem Basis

The hierarchical modular structures are an essential part of the complex biological organisms and play an indispensable role in their evolution. The modularity enhances evolvability allowing recombination of various modules during the evolution process. The evolved hierarchy of modular components bootstraps the evolution process allowing operations over a collection of complex structures rather than basic genes. After that, the neuroevolutionary process does not need to spend time to evolve similar functionality from scratch again. Instead, the ready-to-use modular components can be used as building blocks to produce very complex neural networks.

In this chapter, we implement the solution of the retina problem using the ES-HyperNEAT algorithm. The retina problem is about the simultaneous identification of the valid 2x2 patterns on the left and the right side of the artificial retina, which have resolution 4x2. Thus, the detector ANN must decide if the patterns presented to the left and the right side of the retina are valid for the corresponding side of the retina (left or right).

In the retina problem, the left and the right problem components perfectly separated into different functional units. Thus, to produce successful detector ANN, the neuroevolution process needs to discover the modular structures, separately for the left and the right detection zones.

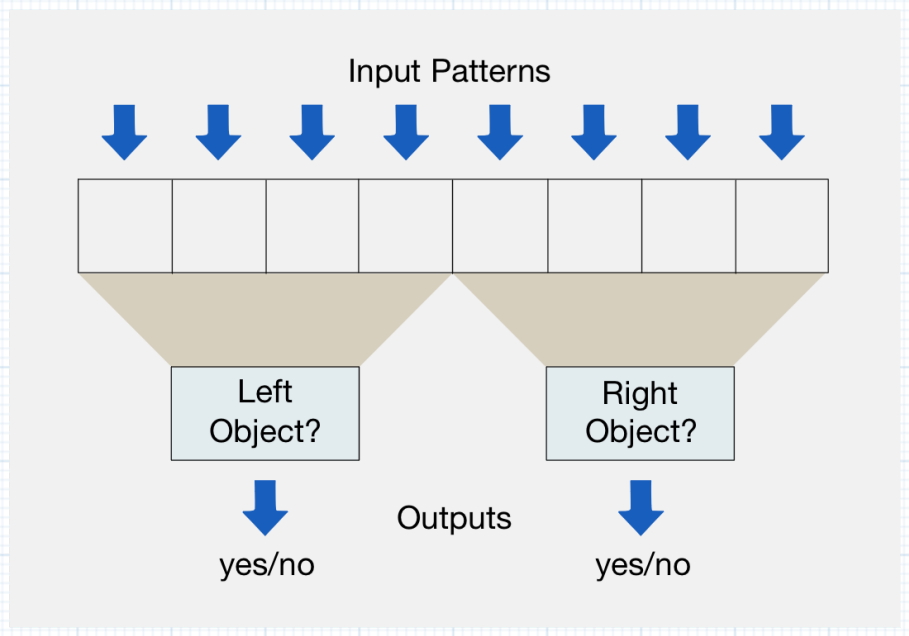
The retina problem scheme is shown in the following image:



[Retina Problem Scheme]

As you can see from the image above, the artificial retina represented as a 2D grid with resolution 4x2 pixels. The values of the two-dimensional array representing patterns drawn on the retina constitute the inputs of the detector ANN. The filled pixels presented in the array with value 1.0 and empty pixels with 0.0. With given resolution, it is possible to draw 16 different 2x2 patterns for the left and the right parts of the retina. Thus, we have the eight valid patterns for the left side and the eight valid patterns for the right side of the retina. Some of the mentioned patterns are valid for both sides of the retina.

The scheme of decision making by the detector ANN in retina problem domain:



[The scheme of decision making by the detector ANN]

The detector ANN has eight inputs to accept input data patters from both sides of the retina and two output nodes. Each of the output nodes produces the value that can be used for classification of the pattern validity at each side of the retina. The first output node is assigned to the left and the second node to the right side of the retina correspondingly. The activation value of the output node that is greater or equal to 0.5 classifies the pattern for the related side of the retina as valid. If the activation value is less than 0.5, the pattern considered as not valid. To even further simplify the detection, we apply rounding to the values of the output nodes according to the rounding scheme given above. Thus, each output node of the detector ANN serves as a binary classifier for the related part of the retina that produces values 0.0 or 1.0 to mark the input pattern as invalid or valid correspondingly.

Objective Function Definition

The task of the detector ANN is to correctly classify the inputs from the left and right sides of the retina as valid or not by producing a vector of the binary outputs having values 0.0 or 1.0. The output vector has length two that is equal to the number of the output nodes.

We can define the detection error as Euclidean distance between the vector with ground truth values and the vector with ANN output values as given by the formula:

$$e^2 = \sum_{i=1}^{2} (a_i - b_i)^2,$$

e^2 is the squared detection error for one trial, a is the vector with detector ANN outputs, and b is the vector with the ground truth values.

At each generation of the evolution, we evaluate each detector ANN (phenotype) against all possible 256 combinations of 4x4 retina patterns, which produced by combining 16 different 2x2 patterns for each side of the retina. Thus, to get a final detection error value for the particular detector ANN, we calculate the sum of 256 error values obtained for each configuration of the retina patterns, as indicated by the formula:

$$\mathscr{E} = \sum_{i=1}^{256} e_i^2,$$

$\mathscr{E}$ is the sum of all errors obtained during 256 trials, e_i^2 is the squared detection error for a particular trial.

The fitness function can be defined as an inverse of the sum of the errors obtained from all 256 trials against all possible retina patterns. The following formula gives it:

$$\mathscr{F} = \frac{1000.0}{1.0 + \mathscr{E}}.$$

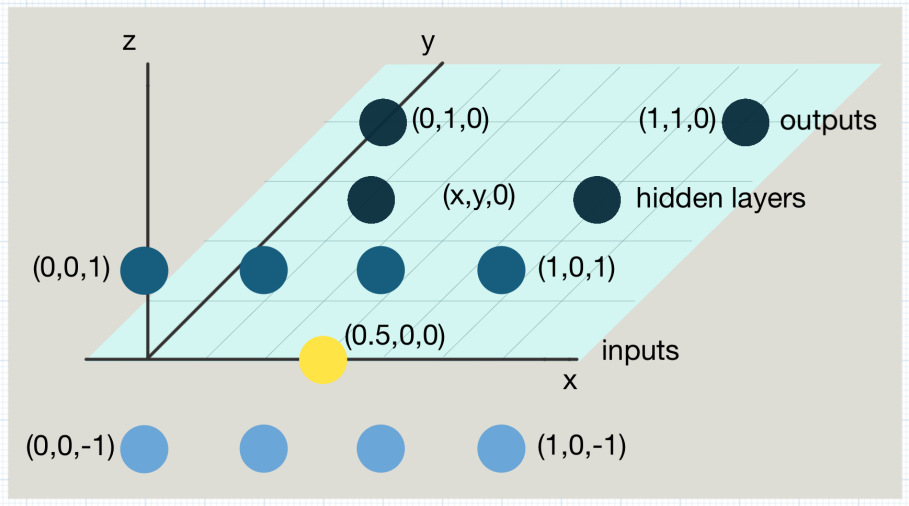
We add 1.0 to the sum of errors ($\mathscr{E}$) in the denominator to avoid division by zero in the case when all trials produce no error. Thus, according to the given fitness function formula, the maximal value of the fitness score in our experiment is 1000.0, which we use as a fitness threshold value later.

Modular Retina Experiment Setup

In this section, we discuss the details of an experiment aimed at creating a successful solver of the modular retina problem. In our experiment, we use this problem as a benchmark to test the ability of the ES-HyperNEAT method to discover modular topologies in the phenotype ANN.

The Initial Substrate Configuration

As described above, the retina has dimension 4x2 with two areas with dimension 2x2 at the left and the right sides. Such particulars of the retina geometry must be represented in the geometry of the initial substrate configuration. In our experiment, we use a three-dimensional substrate, as shown in the following image:



[The initial substrate configuration]

As you can see from the image given above, the input nodes are placed within XZ-pane, which is orthogonal to the XY-plane. They presented in two groups by four nodes to describe the left and right sides of the retina correspondingly. The two output and bias nodes are located within the XY-plane that divides by half the Z-plane with the input nodes. The evolution of the substrate creates new hidden nodes in the same XY-plane where the output nodes are located. The evolved connective CPPN draws the connectivity patterns between all nodes within the substrate. Our ultimate goal is to evolve the CPPN and the substrate configuration, which produce an appropriate modular graph of the detector ANN. This graph should include two modules, each representing an appropriate configuration for the binary classifier, which we discussed above.

Test Environment for the Modular Retina Problem

First, we need to create a test environment that can be used to evaluate the results of the neuroevolution process that is aimed to create successful detector ANN. The test environment should create the data set, which consists of all possible patterns of pixels on the retina. Also, it should provide functions to evaluate the detector ANN against each pattern in the data set. Thus, the test environment can be divided into two major parts:

1. The data structure to hold visual patterns for the left, right, or both sides of the retina
2. The test environment keeping the data set and providing functions for detector ANN evaluation

In the following, we provide a detailed description of each part.

The Visual Object Definition

Each allowed configuration of pixels at the specific part of the retina space can be represented as a separate visual object. The Python class encapsulating related functionality is named *VisualObject* and defined in the ‘retina_experiment.py’ file. It has the following constructor:

```
def __init__(self, configuration, side, size=2):
    self.size = size
    self.side = side
    self.configuration = configuration
    self.data = np.zeros((size, size))

    # Parse configuration
    lines = self.configuration.splitlines()
    for r, line in enumerate(lines):
        chars = line.split(" ")
        for c, ch in enumerate(chars):
            if ch == 'o':
                # pixel is ON
                self.data[r, c] = 1.0
            else:
                # pixel is OFF
                self.data[r, c] = 0.0
```

The constructor above receives the configuration of a particular visual object as a string along with a valid location of this object in retina space. After that, it assigns received parameters to the internal fields and creates the two-dimensional data array holding states of the pixels in the visual object.

The pixels state obtained by parsing the visual object configuration string as follows:

```
    # Parse configuration
    lines = self.configuration.splitlines()
    for r, line in enumerate(lines):
```

```
        chars = line.split(" ")
        for c, ch in enumerate(chars):
            if ch == 'o':
                # pixel is ON
                self.data[r, c] = 1.0
            else:
                # pixel is OFF
                self.data[r, c] = 0.0
```

The visual object configuration string has four characters excluding the line break, which defines the state of the corresponding pixel in the visual object. If the symbol at a specific position in the configuration line is ‘o’, then the pixel at the corresponding position of the visual object is set to ON state, and the value 1.0 saved to the data array at this position.

The Retina Environment Definition

The retina environment creates and holds the data set consisting of all possible visual objects and provides functions for evaluating the fitness of the detector ANN. It has the following major implementation parts.

- The function to create a data set with all visual objects:

```
    def create_data_set(self):
        # set left side objects
        self.visual_objects.append(VisualObject(". .\n. .", side=Side.BOTH))
        self.visual_objects.append(VisualObject(". .\n. o", side=Side.BOTH))
        self.visual_objects.append(VisualObject(". o\n. o", side=Side.LEFT))
        self.visual_objects.append(VisualObject(". o\n. .", side=Side.BOTH))
        self.visual_objects.append(VisualObject(". o\no o", side=Side.LEFT))
        self.visual_objects.append(VisualObject(". .\no .", side=Side.BOTH))
        self.visual_objects.append(VisualObject("o o\n. o", side=Side.LEFT))
        self.visual_objects.append(VisualObject("o .\n. .", side=Side.BOTH))

        # set right side objects
        self.visual_objects.append(VisualObject(". .\n. .", side=Side.BOTH))
        self.visual_objects.append(VisualObject("o .\n. .", side=Side.BOTH))
        self.visual_objects.append(VisualObject("o .\no .", side=Side.RIGHT))
        self.visual_objects.append(VisualObject(". .\no .", side=Side.BOTH))
        self.visual_objects.append(VisualObject("o o\no .", side=Side.RIGHT))
        self.visual_objects.append(VisualObject(". o\n. .", side=Side.BOTH))
        self.visual_objects.append(VisualObject("o .\no o", side=Side.RIGHT))
        self.visual_objects.append(VisualObject(". .\n. o", side=Side.BOTH))
```

The function above creates 16 visual objects that are valid for left, right or both sides of the retina space. The created objects appended to the list of the visual objects defined as a data set for evaluating the fitness of detector ANN produced by the neuroevolution process from the substrate.

- The function to evaluate the detector ANN against two specific visual objects, one for each side of the retina space:

```
    def _evaluate(self, net, left, right, depth, debug=False):

        net.Flush()
        # prepare input
        inputs = left.get_data() + right.get_data()
        inputs.append(0.5) # the bias

        net.Input(inputs)
        # activate
        [net.Activate() for _ in range(depth)]
```

```
    # get outputs
    outputs = net.Output()
    outputs[0] = 1.0 if outputs[0] >= 0.5 else 0.0
    outputs[1] = 1.0 if outputs[1] >= 0.5 else 0.0

    # set ground truth
    left_target = 1.0 if left.side == Side.LEFT or left.side == Side.BOTH else 0.0
    right_target = 1.0 if right.side == Side.RIGHT or right.side == Side.BOTH else 0.0
    targets = [left_target, right_target]

    # find error as a distance between outputs and groud truth
    error = (outputs[0] - targets[0]) * (outputs[0] - targets[0]) + (outputs[1] - targets[1]) * (outputs[1]
- targets[1])

    return error, outputs
```

In the beginning, we prepare inputs for detector ANN in the order as they defined in the substrate configuration:

```
    inputs = left.get_data() + right.get_data()
    inputs.append(0.5) # the bias

    net.Input(inputs)
```

Inputs array starts with left side data and continues with right side data. After that, the bias value appended to the end of the inputs array and the array data supplied as input to the detector ANN.

After a specific number of activations of the detector ANN, the outputs obtained and rounded:

```
    outputs = net.Output()
    outputs[0] = 1.0 if outputs[0] >= 0.5 else 0.0
    outputs[1] = 1.0 if outputs[1] >= 0.5 else 0.0
```

Next, we need to calculate squared detection error, which is Euclidean distance between outputs vector and the vector with ground truth values. Thus, we first create the vector with ground truth values as follows:

```
    left_target = 1.0 if left.side == Side.LEFT or left.side == Side.BOTH else 0.0
    right_target = 1.0 if right.side == Side.RIGHT or right.side == Side.BOTH else 0.0
    targets = [left_target, right_target]
```

The corresponding ground-truth value is set to 1.0 if the visual object is valid for a given side of the retina or both sides. Otherwise, it is set to 0.0 to indicate the wrong visual object position. The squared detection error calculated as follows:

```
    error = (outputs[0] - targets[0]) * (outputs[0] - targets[0]) + \
            (outputs[1] - targets[1]) * (outputs[1] - targets[1])
```

The function returns the detection error and the outputs from detector ANN.

> i: For complete implementation details refer to ‘retina_environment.py’ at https://github.com/PacktPublishing/Hands-on-Neuroevolution-with-Python/blob/master/Chapter8/retina_environment.py

Experiment Runner

To solve the modular retina problem, we need to use a Python library that provides an implementation of the ES-HyperNEAT algorithm. From the previous chapter, you already familiar

with the MultiNEAT Python library, which also has an implementation of the ES-HyperNEAT algorithm. Thus, we can use this library to create a retina experiment runner implementation.

Further, we discuss the essential components of the implementation.

> i: For full implementation details refer to 'retina_experiment.py' at https://github.com/PacktPublishing/Hands-on-Neuroevolution-with-Python/blob/master/Chapter8/retina_experiment.py

---

The Experiment Runner Function

The run_experiment function allows running the experiment using provided hyper-parameters and an initialized test environment to evaluate the discovered detector ANNs against possible retina configurations. The function implementation has the following significant parts.

1. Initialization of the population of initial CPPN genomes:

```
seed = 1569777981
# Create substrate
substrate = create_substrate()
# Create CPPN genome and population
g = NEAT.Genome(0,
                substrate.GetMinCPPNInputs(),
                2, # hidden units
                substrate.GetMinCPPNOutputs(),
                False,
                NEAT.ActivationFunction.TANH,
                NEAT.ActivationFunction.SIGNED_GAUSS, # The initial activation type for hidden
                1, # hidden layers seed
                params,
                1) # one hidden layer
pop = NEAT.Population(g, params, True, 1.0, seed)
pop.RNG.Seed(seed)
```

In the beginning, the code above sets the random seed value to the one that we found to be useful to generate successful solutions. After that, we create the substrate configuration that is suitable for the retina experiment taking into account the geometry of the retina space.

Next, we create the initial CPPN genome, using the substrate configuration we already have. The CPPN genome needs to have the number of input and output nodes that is compatible with the substrate configuration. Also, we decided to seed the initial CPPN genome with two hidden nodes having Gaussian activation function to boost the neuroevolution process in the right direction. The Gaussian hidden nodes allow starting the neuroevolution search with a bias toward producing particular detector ANN topologies. With such hidden nodes, we introduce to the connectivity patters of the substrate the principle of symmetry, which is precisely what we are expecting to achieve in the topology of the successful detector ANN. For the retina problem, we need to discover a symmetrical detector ANN configuration incorporating the two symmetrical classifier modules.

2. Running the neuroevolution process for a specified number of generations:

```
# Run for up to N generations.
start_time = time.time()
best_genome_ser = None
best_ever_goal_fitness = 0
best_id = -1
solution_found = False

stats = Statistics()
```

```
    for generation in range(n_generations):
        print("\n****** Generation: %d ******\n" % generation)
        gen_time = time.time()
        # get list of current genomes
        genomes = NEAT.GetGenomeList(pop)

        # evaluate genomes
        genome, fitness, errors = eval_genomes(genomes, rt_environment=rt_environment,
                                               substrate=substrate, params=params)

        stats.post_evaluate(max_fitness=fitness, errors=errors)
        solution_found = fitness >= FITNESS_THRESHOLD
        # store the best genome
        if solution_found or best_ever_goal_fitness < fitness:
            best_genome_ser = pickle.dumps(genome) # dump to pickle to freeze the genome state
            best_ever_goal_fitness = fitness
            best_id = genome.GetID()

        if solution_found:
            print('Solution found at generation: %d, best fitness: %f, species count: %d' %
(generation, fitness, len(pop.Species)))
            break

        # advance to the next generation
        pop.Epoch()

        # print statistics
        gen_elapsed_time = time.time() - gen_time
        print("Best fitness: %f, genome ID: %d" % (fitness, best_id))
        print("Species count: %d" % len(pop.Species))
        print("Generation elapsed time: %.3f sec" % (gen_elapsed_time))
        print("Best fitness ever: %f, genome ID: %d" % (best_ever_goal_fitness, best_id))
```

We start with the creation of the intermediary variables to hold the execution results along with the statistics collector. After that, we run the evolution loop for a given number of generations:

```
    start_time = time.time()
    best_genome_ser = None
    best_ever_goal_fitness = 0
    best_id = -1
    solution_found = False

    stats = Statistics()
```

Inside the evolution loop, we get the list of genomes belonging to the current population and evaluate it against the test environment as follows:

```
        # get list of current genomes
        genomes = NEAT.GetGenomeList(pop)

        # evaluate genomes
        genome, fitness, errors = eval_genomes(genomes, rt_environment=rt_environment,
                                               substrate=substrate, params=params)

        stats.post_evaluate(max_fitness=fitness, errors=errors)
        solution_found = fitness >= FITNESS_THRESHOLD
```

The eval_genomes function returns a tuple that has the following components: the best fit genome, the highest fitness score among all evaluated genomes, and the list or detection error for each evaluated genome. We save the appropriate parameters into statistics collector and evaluate

obtained fitness score against the search termination criterion, which defined as FITNESS_THRESHOLD constant having value 1000.0. The evolutionary search terminates as successful if the best fitness score in population is greater or equal to the FITNESS_THRESHOLD value.

In case the successful solution was found, or the current best fitness score of the population is higher than the maximal fitness score ever achieved, we save the best CPPN genome and current fitness score as follows:

```
    if solution_found or best_ever_goal_fitness < fitness:
        best_genome_ser = pickle.dumps(genome) # dump to pickle to freeze the genome state
        best_ever_goal_fitness = fitness
        best_id = genome.GetID()
```

After that, if the value of the solution_found variable was set to True, we terminate the evolution loop:

```
    if solution_found:
        print('Solution found at generation: %d, best fitness: %f, species count: %d' %
(generation, fitness, len(pop.Species)))
        break
```

If the evolution failed to produce a successful solution, we print the statistics for the current generation and move to the next epoch:

```
    # advance to the next generation
    pop.Epoch()

    # print statistics
    gen_elapsed_time = time.time() - gen_time
    print("Best fitness: %f, genome ID: %d" % (fitness, best_id))
    print("Species count: %d" % len(pop.Species))
    print("Generation elapsed time: %.3f sec" % (gen_elapsed_time))
    print("Best fitness ever: %f, genome ID: %d" % (best_ever_goal_fitness, best_id))
```

The rest of the experiment runner code reports the results of the experiment in different formats.

3. We report the experiment results in textual and visual formats using statistics collected in the evolution loop. Furthermore, visualizations also saved into the local file system in the SVG vector format.

```
print("\nBest ever fitness: %f, genome ID: %d" % (best_ever_goal_fitness, best_id))
print("\nTrial elapsed time: %.3f sec" % (elapsed_time))
print("Random seed:", seed)

# Visualize the experiment results
show_results = not silent
if save_results or show_results:
    # Draw CPPN network graph
    net = NEAT.NeuralNetwork()
    best_genome.BuildPhenotype(net)
    visualize.draw_net(net, view=False, node_names=None, filename="cppn_graph.svg",
directory=trial_out_dir, fmt='svg')
    print("\nCPPN nodes: %d, connections: %d" % (len(net.neurons), len(net.connections)))

     # Draw the substrate network graph
    net = NEAT.NeuralNetwork()
    best_genome.BuildESHyperNEATPhenotype(net, substrate, params)
    visualize.draw_net(net, view=False, node_names=None, filename="substrate_graph.svg",
directory=trial_out_dir, fmt='svg')
```

```
        print("\nSubstrate nodes: %d, connections: %d" % (len(net.neurons), len(net.connections)))
        inputs = net.NumInputs()
        outputs = net.NumOutputs()
        hidden = len(net.neurons) - net.NumInputs() - net.NumOutputs()
        print("\n\tinputs: %d, outputs: %d, hidden: %d" % (inputs, outputs, hidden))

        # Test against random retina configuration
        l_index = random.randint(0, 15)
        r_index = random.randint(0, 15)
        left = rt_environment.visual_objects[l_index]
        right = rt_environment.visual_objects[r_index]
        err, outputs = rt_environment._evaluate(net, left, right, 3)
        print("Test evaluation error: %f" % err)
        print("Left flag: %f, pattern: %s" % (outputs[0], left))
        print("Right flag: %f, pattern: %s" % (outputs[1], right))

        # Test against all visual objects
        fitness, avg_error, total_count, false_detetctions = rt_environment.evaluate_net(net,
debug=True)
        print("Test evaluation against full data set [%d], fitness: %f, average error: %f, false
detections: %f" %
              (total_count, fitness, avg_error, false_detetctions))

        # Visualize statistics
        visualize.plot_stats(stats, ylog=False, view=show_results, filename=os.path.join(trial_out_dir,
'avg_fitness.svg'))
```

The first three lines of the code above print the general statistics about experiment execution, such as the highest fitness score achieved, the time elapsed for experiment execution, and the random generator seed value.

The next part of the code is about visualization of the experiment results, which is the most informative part, that you should pay great attention to. We start with visualization of the CPPN network that we create from the best genome found during the evolution:

```
        # Draw CPPN network graph
        net = NEAT.NeuralNetwork()
        best_genome.BuildPhenotype(net)
        visualize.draw_net(net, view=False, node_names=None, filename="cppn_graph.svg",
directory=trial_out_dir, fmt='svg')
        print("\nCPPN nodes: %d, connections: %d" % (len(net.neurons), len(net.connections)))
```

After that, we perform visualization of the detector ANN topology that is created using the best CPPN genome and the retina substrate:

```
        net = NEAT.NeuralNetwork()
        best_genome.BuildESHyperNEATPhenotype(net, substrate, params)
        visualize.draw_net(net, view=False, node_names=None, filename="substrate_graph.svg",
directory=trial_out_dir, fmt='svg')
        print("\nSubstrate nodes: %d, connections: %d" % (len(net.neurons), len(net.connections)))
        inputs = net.NumInputs()
        outputs = net.NumOutputs()
        hidden = len(net.neurons) - net.NumInputs() - net.NumOutputs()
        print("\n\tinputs: %d, outputs: %d, hidden: %d" % (inputs, outputs, hidden))
```

Also, we print the results of the evaluation of detector ANN created by the code above against a full data set and two randomly selected visual objects:

```
        # Test against random retina configuration
        l_index = random.randint(0, 15)
```

```
        r_index = random.randint(0, 15)
        left = rt_environment.visual_objects[l_index]
        right = rt_environment.visual_objects[r_index]
        err, outputs = rt_environment._evaluate(net, left, right, 3)
        print("Test evaluation error: %f" % err)
        print("Left flag: %f, pattern: %s" % (outputs[0], left))
        print("Right flag: %f, pattern: %s" % (outputs[1], right))

        # Test against all visual objects
        fitness, avg_error, total_count, false_detetctions = rt_environment.evaluate_net(net,
debug=True)
        print("Test evaluation against full data set [%d], fitness: %f, average error: %f, false
detections: %f" %
              (total_count, fitness, avg_error, false_detetctions))
```

Finally, we render the statistics data collected during the experiment as follows:

```
        # Visualize statistics
        visualize.plot_stats(stats, ylog=False, view=show_results, filename=os.path.join(trial_out_dir,
'avg_fitness.svg'))
```

All visualization plots mentioned above can be found after experiment execution in the trial_out_dir directory of the local file system.

The Substrate Builder Function

The ES-HyperNEAT method allows running the neuroevolution process, which includes the evolution of the CPPN genomes along with the evolution of the substrate configuration. However, even though the substrate is evolving during evolution, it is incredibly beneficial to start with the appropriate initial substrate configuration. This configuration should correspond to the geometry of the problem space.

For the retina experiment, the appropriate substrate configuration created as follows:

```
def create_substrate():
    # The input layer
    x_space = np.linspace(-1.0, 1.0, num=4)
    inputs = [
        (x_space[0], 0.0, 1.0), (x_space[1], 0.0, 1.0), (x_space[0], 0.0, -1.0), (x_space[1], 0.0, -1.0), #
the left side
        (x_space[2], 0.0, 1.0), (x_space[3], 0.0, 1.0), (x_space[2], 0.0, -1.0), (x_space[3], 0.0, -1.0),  #
the right side
        (0,0,0) # the bias
        ]
    # The output layer
    outputs = [(-1.0, 1.0, 0.0), (1.0, 1.0, 0.0)]

    substrate = NEAT.Substrate( inputs,
                                [], # hidden
                                outputs)

    # Allow connections: input-to-hidden, hidden-to-output, and hidden-to-hidden
    substrate.m_allow_input_hidden_links = True
    substrate.m_allow_hidden_output_links = True
    substrate.m_allow_hidden_hidden_links = True

    substrate.m_allow_input_output_links = False
    substrate.m_allow_output_hidden_links = False
    substrate.m_allow_output_output_links = False
    substrate.m_allow_looped_hidden_links = False
```

```
    substrate.m_allow_looped_output_links = False

    substrate.m_hidden_nodes_activation = NEAT.ActivationFunction.SIGNED_SIGMOID
    substrate.m_output_nodes_activation = NEAT.ActivationFunction.UNSIGNED_SIGMOID

    substrate.m_with_distance = True # send connection length to the CPPN as a parameter
    substrate.m_max_weight_and_bias = 8.0

    return substrate
```

First, we create the configuration of the input layer of the substrate. As you remember, from the section where we discussed the initial substrate configuration, the eight nodes of the input layer placed within XZ-pane, which is orthogonal to the XY-plane. Furthermore, to reflect the geometry of the retina space, the left object's nodes need to be placed at the left side, and the right object's nodes at the right side of the plane correspondingly. The bias node should be located at the center of the input nodes plane. Thus, the input layer created as following:

```
    # The input layer
    x_space = np.linspace(-1.0, 1.0, num=4)
    inputs = [
        (x_space[0], 0.0, 1.0), (x_space[1], 0.0, 1.0), (x_space[0], 0.0, -1.0), (x_space[1], 0.0, -1.0), #
the left side
        (x_space[2], 0.0, 1.0), (x_space[3], 0.0, 1.0), (x_space[2], 0.0, -1.0), (x_space[3], 0.0, -1.0),  #
the right side
        (0,0,0) # the bias
        ]
```

The two output nodes located within the XY-plane, which is orthogonal to the inputs plane. Such a substrate configuration allows natural substrate evolution by placing discovered hidden nodes within the XY-plane. The output layer created as follows:

```
    # The output layer
    outputs = [(-1.0, 1.0, 0.0), (1.0, 1.0, 0.0)]
```

Next, we define the general substrate configuration parameters as follows:

```
    # Allow connections: input-to-hidden, hidden-to-output, and hidden-to-hidden
    substrate.m_allow_input_hidden_links = True
    substrate.m_allow_hidden_output_links = True
    substrate.m_allow_hidden_hidden_links = True

    substrate.m_allow_input_output_links = False
    substrate.m_allow_output_hidden_links = False
    substrate.m_allow_output_output_links = False
    substrate.m_allow_looped_hidden_links = False
    substrate.m_allow_looped_output_links = False

    substrate.m_hidden_nodes_activation = NEAT.ActivationFunction.SIGNED_SIGMOID
    substrate.m_output_nodes_activation = NEAT.ActivationFunction.UNSIGNED_SIGMOID

    substrate.m_with_distance = True # send connection length to the CPPN as a parameter
    substrate.m_max_weight_and_bias = 8.0
```

We allow the substrate to have connections as input to hidden, hidden to hidden, and hidden to output nodes. We specify that hidden nodes should use signed sigmoid activation function, while output nodes should use unsigned sigmoid activation function. We choose the unsigned sigmoid activation for the output nodes to have detector ANN output values in the range [0,1].

Fitness Evaluation

The neuroevolution process requires a means to evaluate the fitness of the genomes population at each generation of evolution. The fitness evaluation in our experiment consists of two parts, which we discuss below.

- We implemented the fitness evaluation of the overall population of the CPPN genomes in the eval_genomes function, which has the following definition:

```
def eval_genomes(genomes, substrate, rt_environment, params):
    best_genome = None
    max_fitness = 0
    errors = []
    for genome in genomes:
        fitness, error, total_count, false_detetctions = eval_individual(genome, substrate,
rt_environment, params)
        genome.SetFitness(fitness)
        errors.append(error)

        if fitness > max_fitness:
            max_fitness = fitness
            best_genome = genome

    return best_genome, max_fitness, errors
```

The eval_genomes function takes as parameters the list of CPPN genomes from the current population, the substrate configuration, the initialized test environment, and the ES-HyperNEAT parameters.

At the beginning of the code, we create an intermediary object to collect evaluation results of each specific genome:

```
    best_genome = None
    max_fitness = 0
    errors = []
```

After that, we start the loop that iterates over all genomes and evaluates each genome against a given test environment:

```
    for genome in genomes:
        fitness, error, total_count, false_detetctions = eval_individual(genome, substrate,
rt_environment, params)
        genome.SetFitness(fitness)
        errors.append(error)

        if fitness > max_fitness:
            max_fitness = fitness
            best_genome = genome
```

Finally, the function returns the evaluation results as a tuple that includes the best genome, the highest fitness score, and the list of all detection errors per each evaluated genome.

- The eval_individual function, which invoked to evaluate the fitness of the individual genome defined as follows:

```
def eval_individual(genome, substrate, rt_environment, params):
    # Create ANN from provided CPPN genome and substrate
    net = NEAT.NeuralNetwork()
    genome.BuildESHyperNEATPhenotype(net, substrate, params)
```

```
    fitness, dist, total_count, false_detetctions = rt_environment.evaluate_net(net,
max_fitness=MAX_FITNESS)
    return fitness, dist, total_count, false_detetctions
```

It takes as parameters the CPPN genome to be evaluated, the substrate configuration, the test environment, and the ES-HyperNEAT hyper-parameters. Using the provided parameters, we create the neural network configuration of the detector ANN and evaluate it against a given test environment. The function then returns the evaluation result.

Modular Retina Experiment

Now, we are ready to start experimenting against the test environment that simulates the modular retina problem space. In the next subsections, you will learn how to select appropriate hyper-parameters as well as how to set up the environment and run the experiment. After that, we discuss the experiment results.

Hyper-Parameters Selection

The hyper-parameters are defined as a Parameters Python class, and the MultiNEAT library refers to it for the necessary configuration options. In the source code of the experiment runner script, we define a specialized function create_hyperparameters, which encapsulates the logic of the hyper-parameters initialization. Hereafter, we describe the most critical hyper-parameters and the reasons for choosing specific values.

1. We decided to use the big enough size of the CPPN genomes population. It is done to intensify the evolution by providing from beginning a vast space of variants for the solution search. The size of the population defined as follows:

   ```
   params.PopulationSize = 300
   ```

2. Next, we define the number of species to be kept during evolution in the range [5,15] and set the species stagnation to 100 generations. This configuration allows us to have a healthy diversity among species and keep them alive long enough to produce a solution we are looking for:

   ```
   params.SpeciesMaxStagnation = 100
   params.MinSpecies = 5
   params.MaxSpecies = 15
   ```

3. We are interested in producing the extra compact configuration of CPPN genomes. Thus, we have very small values of probabilities that control how often new nodes and connections will be introduced into the genome:

   ```
   params.MutateAddLinkProb = 0.03
   params.MutateAddNeuronProb = 0.03
   ```

4. The ES-HyperNEAT method is an extension of the HyperNEAT method. Thus, during the evolution, it changes the types of activation functions at the hidden and output nodes. In this experiment, to produce appropriate substrate configurations, we are interested in the following activation types selected with equal probability:

   ```
   params.ActivationFunction_SignedGauss_Prob = 1.0
   params.ActivationFunction_SignedStep_Prob = 1.0
   params.ActivationFunction_Linear_Prob = 1.0
   params.ActivationFunction_SignedSine_Prob = 1.0
   params.ActivationFunction_SignedSigmoid_Prob = 1.0
   ```

5. Finally, we define the ES-HyperNEAT specific hyper-parameters, which controls how substrate evolves. The following hyper-parameters control the dynamics of the creation of nodes and connections within a substrate during the evolution:

```
params.DivisionThreshold = 0.5
params.VarianceThreshold = 0.03
```

The *params.DivisionThreshold* controls how many new nodes and connections are introduced into the substrate at each generation of evolution. The *params.VarianceThreshold* determines many nodes and connections that remain in the substrate from the new ones created above.

Working Environment Setup

In this experiment, we use the MultiNEAT Python library, which provides the implementation of the ES-HyperNEAT algorithm. Thus, we need to create an appropriate Python environment, which includes the MultiNEAT Python library and all necessary dependencies. This can be done using Anaconda Distribution by executing the following commands in the command line:

```
$ conda create –name rt_multineat python=3.5
$ conda activate vd_multineat
$ conda install -c conda-forge multineat
$ conda install matplotlib
$ conda install -c anaconda seaborn
$ conda install graphviz
$ conda install python-graphviz
```

The commands above create and activate the ‘rt_multineat’ virtual environment with Python 3.5. After that, it installs the MultiNEAT Python library with the latest version along with dependencies that used by our code for results visualization.

Running the Modular Retina Experiment

At this stage, we already have the experiment runner script fully defined in the ‘retina_experiment.py’ Python script. You can start the experiment by cloning the corresponding Git repository and running the script with the following commands:

```
$ git clone https://github.com/PacktPublishing/Hands-on-Neuroevolution-with-Python.git
$ cd Hands-on-Neuroevolution-with-Python/Chapter8
$ python retina_experiment.py -t 1 -g 1000
```

> i: Do not forget to activate the appropriate virtual environment with the following command:
> **conda activate rt_multineat**

The command above starts one trial of the experiment for 1000 generations of evolution. After a particular number of generations, the successful solution become found, and you can see the following output in the console:

```
****** Generation: 949 ******

Solution found at generation: 949, best fitness: 1000.000000, species count: 6

Best ever fitness: 1000.000000, genome ID: 284698

Trial elapsed time: 1332.576 sec
```

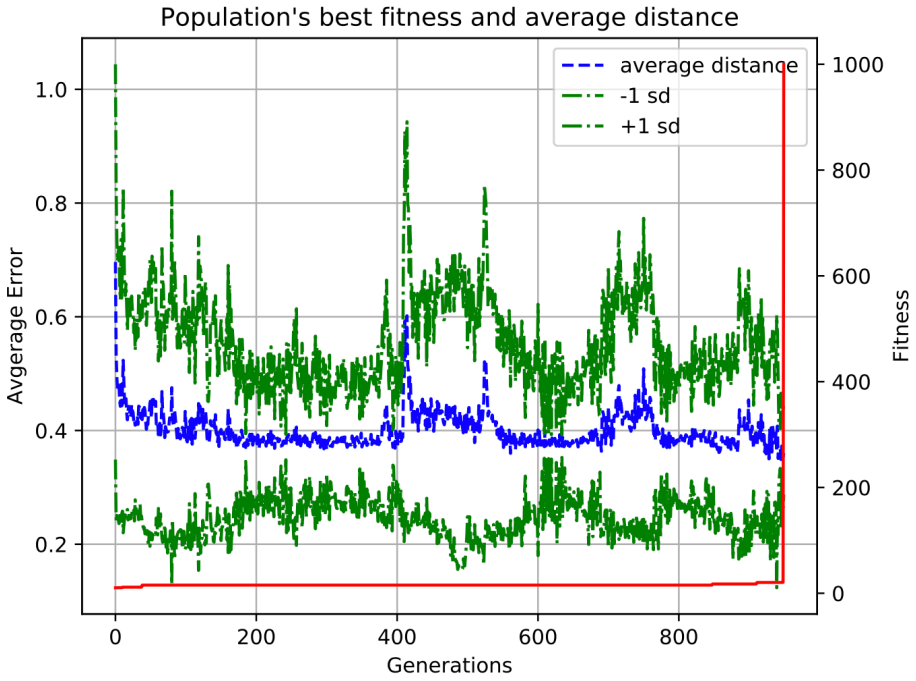
```
Random seed: 1569777981

CPPN nodes: 21, connections: 22

Substrate nodes: 15, connections: 28
```

As you can see from the output above, the successful solution was found at generation 949. It was produced by the CPPN genome having 21 nodes and 22 connections among them. At the same time, the substrate that determines the topology of the detector ANN has 15 nodes and 28 connections between them. The successful solution was produced using random seed value 1569777981. Using other random seed values may fail to produce successful solutions, or it will require much more generations of evolution.

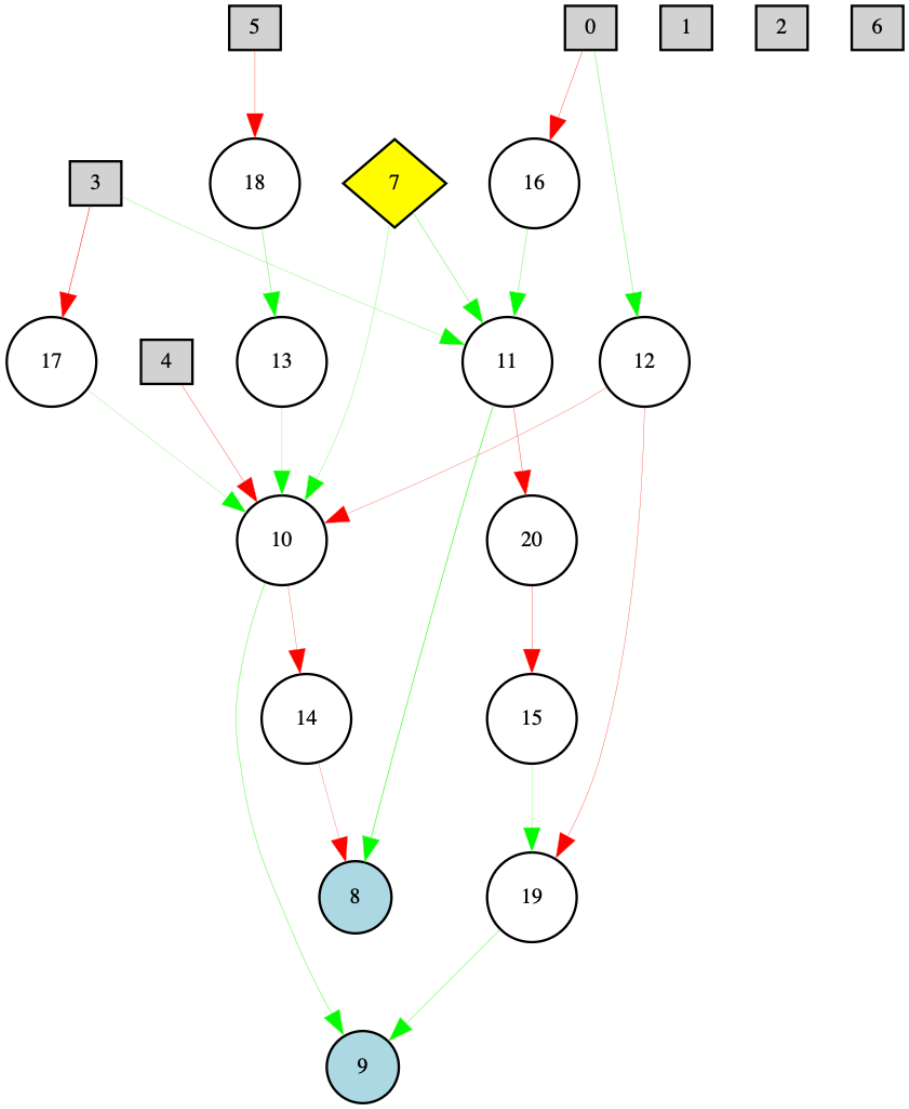
Next, it is interesting to look at the plot with average fitness and error during the evolution:



The average fitness and error per generation.

You can see from the plot above that most of the evolution generations the fitness score was very small (about 20), but suddenly successful CPPN genome was found that produced immediate evolutionary leap just in one generation.

The configuration of the successful CPPN genome is shown in the following image:



The CPPN phenotype graph of the successful genome

The plot above is extremely interesting because, as you can see, the configuration of the successful CPPN genome does not use all the available inputs (grey squares) to produce outputs. Moreover, even more confounding is that it uses only the X-coordinate of the input (node #0) and the Y-coordinate of the hidden (node #3) substrate nodes when deciding about exposing a connection between these substrate nodes. At the same time, both X and Y coordinates of the substrate output nodes involved in the decision-making process (nodes #4 and #5).

When you look at the initial substrate configuration, which we presented earlier, you will see that mentioned above peculiarities, are fully substantiated by the substrate topology. We placed the input nodes within XZ-plane. Thus, the Y-coordinate is not critical for them at all. At the same time, the hidden nodes located within XY-plane with Y-coordinate determining the distance from the inputs plane. Finally, the output nodes are also located within XY-plane. Their X-coordinate determines the side of the retina, to which each output node relates. Thus, for the output nodes, it is natural that both X and Y-coordinates are included.

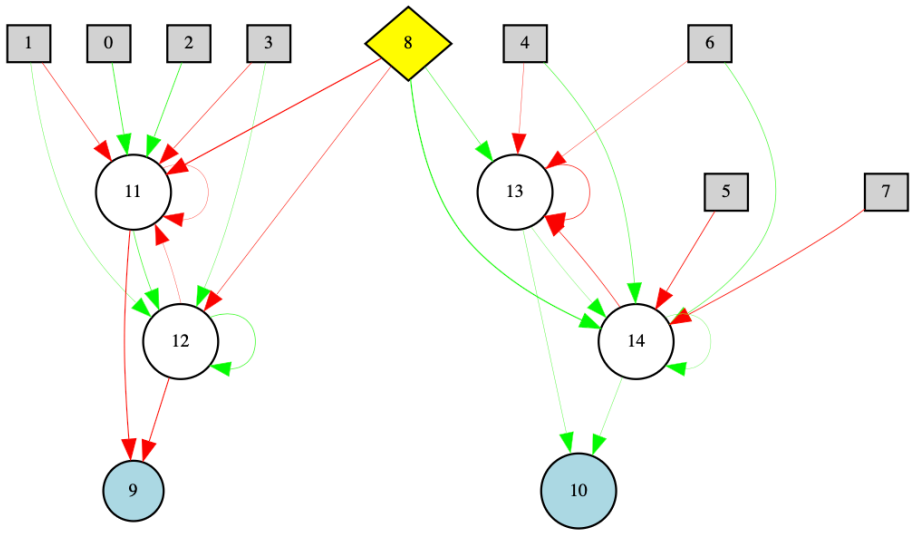
In the CPPN phenotype plot, input nodes marked with squares, output nodes with filled circles, the bias node as a diamond, and the hidden nodes as empty circles.

Two output nodes of the CPPN has the following meaning:
- The first node (8) provides the weight of the connection
- The second node (9) determines whether the connection expressed or not.

The CPPN's input nodes defined as the following:

- First two nodes (0 and 1) to set the point coordinates (x, y) in the input layer of the substrate
- Next two nodes (2 and 3) to set the point coordinates (x, y) in the hidden layer of the substrate
- Next two nodes (4 and 5) to set the point coordinates (x, y) in the output layer of the substrate
- Last node (6) to set the Euclidean distance of the point in the input layer from the origin of the coordinates

However, the most exciting part of the experiment results you can see in the following graph with the configuration of the successful detector ANN:



The configuration of the detector ANN

As in the previous plot, we mark the input nodes with squares, the output nodes with filled circles, the bias node as a diamond, and the hidden nodes as empty circles.

As you can see, we have two clearly separated modular structures at the left and the right sides of the graph. Each module is connected to the corresponding inputs from the left (nodes #0, #1, #2, #3) and the right (nodes #4, #5, #6, #7) sides of the retina. Both modules have the same number of hidden nodes, which are connected to the corresponding output nodes: node #9 for the left side and node #10 for the right side of the retina. Also, you can see that connectivity patterns in the left and right modules are similar. The hidden node #11 at the left has similar connection patterns to the node #14 at the right as well as node #12 to node # 13.

It is just amazing how the stochastic evolutionary process was able to discover such a simple and elegant solution. With the results of this experiment, we fully confirmed our hypothesis that the retina problem could be solved by the creation of the modular detector ANN topologies.

Exercises

1. Try to run an experiment with different values of the random seed generator that can be changed in line 101 of the 'retina_experiment.py' script. See if you would be able to find successful solutions with other values.
2. Try to increase the initial population size to 1000 by adjusting the value of the hyper-parameter params.PopulationSize. How this affected the performance of the algorithm?
3. Try to change the number of activation function types used during the evolution by setting to zero the probability of its selection. Especially interesting to see what is happening when you exclude from a selection the ActivationFunction_SignedGauss_Prob, ActivationFunction_SignedStep_Prob activation types.

Summary

In chapter 8, you learned about the neuroevolution method that allows the substrate configuration to evolve during the process. Such an approach frees the human designer from the burden of creating the proper substrate configuration to the smallest details, allowing us to define only the primary outlines. The algorithm will automatically learn the remaining details of the substrate configuration during the evolution.

Also, you learned about the modular ANN structures that can be used to solve various problems, including the modular retina problem. The modular ANN topologies are a very powerful concept that allows the reuse of the successful phenotype ANN module multiple times to build complex hierarchical topology.

Furthermore, you had a chance to hone your skills with the Python programming language by implementing the corresponding solution using the MultiNEAT Python library.

In the next chapter, we will discuss a fascinating concept of coevolution and how it can be used to simultaneously coevolve the solver and the objective function that is used for optimization. We will discuss the method of Solution and Fitness Evolution, and you will learn how to apply it to the modified maze solving experiment.